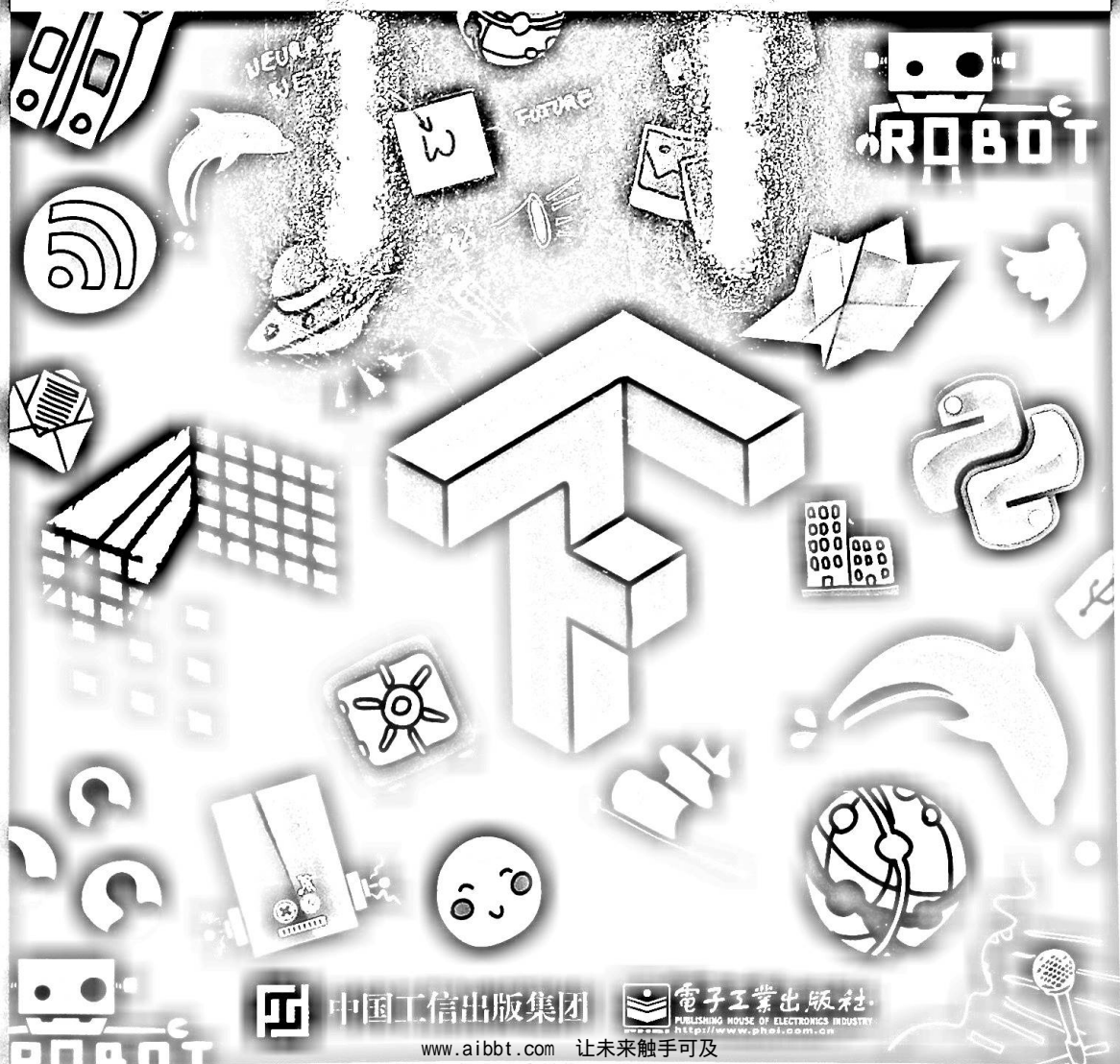


Broadview®
www.broadview.com.cn

深度学习原理与TensorFlow实践

喻俨 莫瑜 主编
王琛 胡振邦 高杰 著



内 容 简 介

本书主要介绍了深度学习的基础原理和 TensorFlow 系统基本使用方法。TensorFlow 是目前机器学习、深度学习领域最优秀的计算系统之一，本书结合实例介绍了使用 TensorFlow 开发机器学习应用的详细方法和步骤。同时，本书着重讲解了用于图像识别的卷积神经网络和用于自然语言处理的循环神经网络的理论知识及其 TensorFlow 实现方法，并结合实际场景和例子描述了深度学习技术的应用范围与效果。

本书非常适合对机器学习、深度学习感兴趣的读者，或是对深度学习理论有所了解，希望尝试更多工程实践的读者，抑或是对工程产品有较多经验，希望学习深度学习理论的读者。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目 (CIP) 数据

深度学习原理与 TensorFlow 实践 / 喻俨，莫瑜主编；王琛，胡振邦，高杰著. —北京：电子工业出版社，2017.6

ISBN 978-7-121-31298-4

I. ①深… II. ①喻… ②莫… ③王… ④胡… ⑤高… III. ①人工智能—算法—研究
IV. ①TP18

中国版本图书馆 CIP 数据核字(2017)第 071463 号

策划编辑：刘 皎

责任编辑：刘 皎

特约编辑：顾慧芳

印 刷：北京季蜂印刷有限公司

装 订：北京季蜂印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：720×1000 1/16 印张：19 字数：364.8 千字

版 次：2017 年 6 月第 1 版

印 次：2017 年 6 月第 1 次印刷

定 价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888，88258888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。



好评袭来

可能有的人还没有觉察到，当前正是新的一场技术革命爆发的起始点。人工智能时代从“即将来临”已经变成了“正在来临”。推动这场技术革命的正是深度学习技术的发展，Google 的围棋算法 AlphaGo 战胜了李世石，Google 的深度学习框架 TensorFlow 也迅速风靡业界，一跃成为最活跃的深度学习框架。

本书基于作者们使用 TensorFlow 的一手实践，由浅入深地介绍了 TensorFlow 架构和其上的各种深度神经网络算法实现，并给出实际的例子，非常适合 AI 爱好者学习，可以较全面地掌握深度学习的知识，并具备实战的能力。未来的 AI 时代里，深度学习技术将成为程序员重要的基础能力，向所有意识到这一点的人推荐此书！

——爱因互动创始人&CTO 洪强宁

TensorFlow 的出现和成熟，改变了深度学习的入门和深造路径。今天我们完全有可能从具体需求出发，以实践主导，比较容易地入门这一前沿人工智能技术。但是要超越写写例子、做做 Demo 的层次，创造性地解决新问题，必须在理论上达到一定的理解高度。本书就是沿着这样一个思路展开的，本书作者开辟了一条由实践主导、兼顾理论的深度学习成功之路，而且语言生动，行文细腻，交代清晰，对后来的学习者是一份难得的指南。

——AI100 联合创始人 孟岩

本书深入浅出地介绍了 TensorFlow 的技术架构以及深度学习领域常见的网络结构和相关理论,并结合图像、文本分析处理等多个实用的具体实例演示了如何使用 TensorFlow 实战深度学习开发,是一本内容翔实的 TensorFlow 开发指导书,强烈推荐!

——东方网力科技股份有限公司 CTO 蒋宗文

随着深度学习在语音、图像和自然语言理解等领域取得了巨大的技术进步,对于初学者而言,一本深入浅出、通俗易懂、融合基础理论和实战的入门书非常重要。

近年来,TensorFlow 广受业界追捧。因此,本书以此平台作为介绍深度学习的媒介,有利于读者快速地运用所学知识,融合到工程实践、科研或系统研发任务中去。

另外,本书的一个主要特点还在于理论和实践的有机结合。本书较全面地介绍了深度学习的基础理论知识,还通过丰富的实例及代码为读者提供了在 TensorFlow 平台上进行实践的机会——这大大增加了培养读者学习兴趣和实战经验的可能性。书中给出的实例涉及图像处理、自然语言理解、对话系统、看图说话等。

本书不仅适合广大本科同学以及研究生入门学习使用,也可以作为经验丰富的工程师或者研究人员的案头参考。

衷心祝愿此书能够成为深度学习爱好者的良师益友!

——徐金安 博士

毫无疑问,深度学习是近年来人工智能和机器学习领域最热门的技术,在很多应用领域发挥着革命性的作用。同样毫无疑问,深度学习得以广泛应用的重要原因之一,是很多公司与学术机构推出了高效可用的深度学习开源框架,使人人都能够快速构建自己的深度学习模型。在众多深度学习框架中,TensorFlow 出自深度学习重要推手的 Google 公司,甫一问世就得到大家密切关注。经过 Google 的几次密级改版,现在 TensorFlow 已经快速成长为深度学习的首选开发平台。本书系统介绍了深度学习的基本思想和 TensorFlow 的实现方法,是深度学习快速上手和入门的好书。

——清华大学计算机系助理教授 刘知远



作者简介

喻俨，百纳信息（海豚浏览器）研发副总裁。2007 年加入微软亚洲工程院，2011 年加入百纳信息负责海外业务线，从 0 到 1 做过多个项目，现致力于 AI 和大数据产品的研究与应用。

莫瑜，先后任职于微软和海豚浏览器，从事搜索引擎、音乐检索/哼唱搜索、内容分发推荐算法和对话机器人技术研发。长期以来持续关注和实践大规模数据算法性能优化、搜索引擎、推荐系统和人工智能技术。

王琛，英国爱丁堡大学人工智能专业硕士，现为百纳信息技术有限公司人工智能方向负责人。早年参加过信息学奥林匹克竞赛获得河北省第一名、全国三等奖，并保送进入中山大学。大学期间，在 ACM 竞赛上也屡获佳绩。硕士毕业后就职于百度基础架构部，参与大数据平台研发工作，对大数据分析处理、分布式系统架构等方面都有比较深刻的理解。2014 年加入百纳，负责多个项目的研发，自 2016 年起负责人工智能方向的探索。

胡振邦，拥有博士学位，百纳信息技术有限公司高级算法研究员，毕业于中国地质大学计算机学院地学信息工程专业。读博期间，参与了关于遥感卫星图像识别分析的 863 项目，并且是主要的研发人员。毕业以来，一直从事图像识别方面的算法研发工作，主要方向包括目标检测、图文检索、图像分类与验证等，在图像处理、计算机视觉等方面都有深厚的积累和经验。

高杰，是一位 1980 年出生于苏北的“爱学习、能折腾、有情怀”的大叔。毕业于扬州中学特招班，1998 年入学华中科技大学机械系，兼修管理、会计，自学计算机，2003 年考入南京大学软件学院，曾任德国西门子内部 SAP 咨询师，还在中银国际 TMT 投行、金山软件集团投资部任过职，2015 年与合伙人联合创立了图灵科技集团，与华尔街顶尖交易团队一起致力于量化交易、算法模型和人工智能在金融领域的应用，目前这家公司管理着超过 20 亿元的资产，是细分市场的领先公司。



序

从理论到工程

技术发展的过程就是人类在探索自身创造能力边界的过程,而人工智能无疑是最重要以及影响最深远的领域之一。

AlphaGo 及其马甲 Master 在围棋领域大胜人类顶尖高手之后,在智力分析领域人类优势开始出现裂痕。而在“听说读写”方面,不管是语音识别、语音合成、机器翻译,还是图像识别、物体识别,甚至是自动文章生成、自动曲谱生成、艺术图像合成方面,机器已经开始做得比人类更为强大。深度学习在工程领域的突破,使得“机器学习”走出了实验室,进入到工程领域,人类开始重新审视机器能带来的更多可能性。

正如 2007 年以 iPhone 为代表的智能手机出现,10 年之间已经颠覆了诸多商业领域、影响了人类的生活方式一样,深度学习也必将如此,作为一名技术人,必须理解和跟上行业和时代的变革!

在过去的计算机技术演变过程中,数据主线(展示、逻辑、存储)、架构主线(C/S, B/S, SASS)、语言框架平台主线(语法、库、框架、操作系统、平台)的变迁基本有迹可循,易于举一反三,迁移学习曲线相对平缓。而机器学习的学习曲线相当陡峭,需要同时专注于数据处理、模型构建以及结果优化,颠覆了我们以往对数据处理的理解。作为工程业界人士来说,没有机器学习理论基础的支撑,几乎无法应用相关的工

具；而没有工程实践的尝试，又很难体系化理解理论基础——入门着实不易。

本书的作者为具有多年研究经验的博士和多年业界工程研发经验的团队，他们在工程领域的经验能快速地帮助读者理解 TensorFlow 的基础概念，并以最快速度搭建环境和跑通 Demo。更为重要的是，他们从学术+工程领域的角度，高屋建瓴地拎出了 CNN（卷积神经网络）、RNN（循环神经网络）、CNN+LSTM（Long Short Term 网络）的基本原理，并且结合 CNN 在图像领域处理、RNN 在语义领域处理以及结合 CNN+LSTM 在图像检测和图像摘要生成等基本工程领域的处理，快速地让读者理解深度学习能干什么，如何利用 TensorFlow 快速解决这些问题，让自己的应用插上“人工智能”的翅膀！

人工智能的时代已经开启，唯有快速拥抱变化才能应对变化，希望读者能借这本书建立对机器学习的宏观认识并对之深入理解，跑步进入机器学习领域！

刘铁锋

《编程之美》作者

海豚浏览器创始人



前言

创造出具有智能的机器一直是人们梦寐以求的理想。自 20 世纪 50 年代图灵测试被提出以来，人工智能就成为了计算机科学领域中一个极具吸引力的研究方向。近年来，深度学习是机器学习领域中一个非常具有突破性的研究方向，从 AlphaGo 战胜李世石，到 Prisma 运用深度学习技术制作滤镜刷爆全世界的社交网络，深度学习在图像处理、自然语言处理甚至博弈决策等问题上不断取得震惊世人的成绩。

随着科研理论上的不断突破，机器学习基础架构方面也有了长足进步。为了提高科研和应用的开发效率，面向深度学习的开发框架不断涌现，而 TensorFlow 就是其中的佼佼者。依托于 Google 强大的影响力，TensorFlow 一经发布就吸引了整个行业的关注。TensorFlow 自 2015 年年底在 GitHub 开源以来，一直是机器学习、深度学习类别中关注度最高的项目，截至 2016 年年底，已经获得超过 40000 个 Star。同时，在开源社区共同的努力下，基于 TensorFlow 开发的各种算法和应用都在飞速增加。

本书结合基于 TensorFlow 实践的应用代码，介绍了深度学习的基础概念和知识，但需要读者预先掌握一些传统机器学习、神经网络相关方面的知识。同时，本书代码主要基于目前最新的 TensorFlow 1.0 版本，大部分为 Python 代码，需要读者有一定的 Python 语言基础。希望通过本书的介绍，读者可以由浅入深、由理论到实践全面掌握深度学习的基础知识和实践方法。

本书第 1 章介绍了深度学习的由来以及发展趋势，简要说明了人工智能、机器学

习、深度学习等名词概念之间的联系。第 2 章主要介绍了 TensorFlow 系统的基础知识和一些重要概念。第 3 章通过对 Kaggle 竞赛平台上的 Titanic 问题的求解实例，介绍了 TensorFlow 系统的基本用法，并简要介绍了机器学习问题中的一些常用的处理技巧。第 4 章和第 5 章分别介绍了主要应用于图像处理领域的卷积神经网络 CNN 和主要应用于自然语言处理领域的循环神经网络 RNN。其中第 4 章介绍了 CNN 的基本原理和多个经典网络结构，并通过图像风格化的实例展示了 CNN 在更多场景下应用的可能性。第 5 章介绍了 RNN、LSTM 以及它们的多种变种结构，并通过实例介绍了如何构建实用的语言模型和对话机器人。第 6 章介绍了卷积神经网络与循环神经网络的结合，通过图像检测和图像摘要两个问题介绍了 CNN+LSTM 相结合的威力。最后的第 7 章介绍了机器学习中非常重要的损失函数与优化算法在 TensorFlow 中的实现，对实际使用深度学习解决问题都有极大帮助。

在此感谢互联网时代，感谢 Google 的开源精神，让我们可以如此紧跟时代最前沿的技术，也可以为技术的进步做出自己微薄的贡献。还要感谢电子工业出版社刘蛟编辑对新技术的关注和推广，感谢同事、家人、各位好友的支持和帮助，有你们的支持才有此书的出版，不胜感激。

作者

注册博文视点 (www.broadview.com.cn) 用户，享受以下服务：

- 提勘误赚积分：可在【提交勘误】处提交对内容的修改意见，若被采纳将获赠博文视点社区积分（可用来抵扣购买电子书的相应金额）。
- 交流学习：在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者共同交流。

页面入口：<http://www.broadview.com.cn/31298>

二维码入口：



目 录

1	深度学习简介	1
1.1	深度学习介绍.....	1
1.2	深度学习的趋势.....	7
1.3	参考资料.....	10
2	TensorFlow 系统介绍	12
2.1	TensorFlow 诞生的动机.....	12
2.2	TensorFlow 系统简介.....	14
2.3	TensorFlow 基础概念.....	16
2.3.1	计算图.....	16
2.3.2	Session 会话.....	18
2.4	系统架构.....	19
2.5	源码结构.....	21
2.5.1	后端执行引擎.....	22
2.5.2	前端语言接口.....	24
2.6	小结.....	24
2.7	参考资料.....	25
3	Hello TensorFlow	26
3.1	环境准备.....	26

3.1.1	Mac OS 安装	27
3.1.2	Linux GPU 服务器安装	28
3.1.3	常用 Python 库	32
3.2	Titanic 题目实战	34
3.2.1	Kaggle 平台介绍	34
3.2.2	Titanic 题目介绍	35
3.2.3	数据读入及预处理	38
3.2.4	构建计算图	40
3.2.5	构建训练迭代过程	44
3.2.6	执行训练	46
3.2.7	存储和加载模型参数	47
3.2.8	预测测试数据结果	50
3.3	数据挖掘的技巧	51
3.3.1	数据可视化	52
3.3.2	特征工程	54
3.3.3	多种算法模型	57
3.4	TensorBoard 可视化	58
3.4.1	记录事件数据	58
3.4.2	启动 TensorBoard 服务	60
3.5	数据读取	62
3.5.1	数据文件格式	63
3.5.2	TFRecord	63
3.6	SkFlow、TFLearn 与 TF-Slim	67
3.7	小结	69
3.8	参考资料	69
4	CNN “看懂” 世界	71
4.1	图像识别的难题	72
4.2	CNNs 的基本原理	74
4.2.1	卷积的数学意义	75
4.2.2	卷积滤波	77
4.2.3	CNNs 中的卷积层	81
4.2.4	池化 (Pooling)	83
4.2.5	ReLU	84

4.2.6	多层卷积.....	86
4.2.7	Dropout.....	86
4.3	经典 CNN 模型.....	87
4.3.1	AlexNet.....	88
4.3.2	VGGNets.....	95
4.3.3	GoogLeNet & Inception.....	98
4.3.4	ResNets.....	106
4.4	图像风格转换.....	109
4.4.1	量化的风格.....	109
4.4.2	风格的滤镜.....	116
4.5	小结.....	120
4.6	参考资料.....	121

5

RNN “能说会道” 123

5.1	文本理解和文本生成问题.....	124
5.2	标准 RNN 模型.....	128
5.2.1	RNN 模型介绍.....	128
5.2.2	BPTT 算法.....	130
5.2.3	灵活的 RNN 结构.....	132
5.2.4	TensorFlow 实现正弦序列预测.....	135
5.3	LSTM 模型.....	138
5.3.1	长期依赖的难题.....	138
5.3.2	LSTM 基本原理.....	139
5.3.3	TensorFlow 构建 LSTM 模型.....	142
5.4	更多 RNN 的变体.....	144
5.5	语言模型.....	146
5.5.1	NGram 语言模型.....	146
5.5.2	神经网络语言模型.....	148
5.5.3	循环神经网络语言模型.....	150
5.5.4	语言模型也能写代码.....	152
5.5.5	改进方向.....	163
5.6	对话机器人.....	164
5.6.1	对话机器人的发展.....	165
5.6.2	基于 seq2seq 的对话机器人.....	169

5.7	小结.....	181
5.8	参考资料.....	182
6	CNN+LSTM 看图说话	183
6.1	CNN+LSTM 网络模型与图像检测问题.....	184
6.1.1	OverFeat 和 Faster R-CNN 图像检测算法介绍.....	185
6.1.2	遮挡目标图像检测方法.....	187
6.1.3	ReInspect 算法实现及模块说明.....	188
6.1.4	ReInspect 算法的实验数据与结论.....	204
6.2	CNN+LSTM 网络模型与图像摘要问题.....	207
6.2.1	图像摘要问题.....	208
6.2.2	NIC 图像摘要生成算法.....	209
6.2.3	NIC 图像摘要生成算法实现说明.....	214
6.2.4	NIC 算法的实验数据与结论.....	243
6.3	小结.....	249
6.4	参考资料.....	250
7	损失函数与优化算法	253
7.1	目标函数优化策略.....	254
7.1.1	梯度下降算法.....	254
7.1.2	RMSProp 优化算法.....	256
7.1.3	Adam 优化算法.....	257
7.1.4	目标函数优化算法小结.....	258
7.2	类别采样 (Candidate Sampling) 损失函数.....	259
7.2.1	softmax 类别采样损失函数.....	261
7.2.2	噪声对比估计类别采样损失函数.....	281
7.2.3	负样本估计类别采样损失函数.....	286
7.2.4	类别采样 logistic 损失函数.....	286
7.3	小结.....	287
7.4	参考资料.....	288
	结语	289

1

深度学习简介

1.1 深度学习介绍

深度学习是目前机器学习学科发展最蓬勃的分支,也是整个人工智能领域中应用前景最为广阔的技术。在现如今的生活,不管是在 iPhone 上随手调戏 Siri,还是看着 AlphaGo 赢得围棋世界第一的宝座,都让人们真真切切地感受到人工智能已经不再是停留在科幻小说中的幻想,深度学习的时代已经到来了!

人工智能 (Artificial Intelligence, AI) 是计算机科学中的一个分支学科,早在 20 世纪 50 年代就被提出和确立了。著名的“图灵测试”是 AI 发展的终极目标,如果某种机器运行的逻辑程序可以表现出与人类等价或者无法分辨的智能,则认为机器有了思维,能够进行思考。从实用的角度讲, AI 的目标是要让计算机系统能够自动完成那些需要依靠人类智慧才能完成的工作。

在 AI 发展的早期阶段,随着计算机自动化所取得的成功, AI 的主要方法和思路是将人类总结的知识用一系列规范的、形式化的数学规则来表示,然后通过自动化的程序代替人类处理问题。以知识为基础的专家系统 (knowledge-based expert system) 就是这方面的典型代表,它将某个领域中人类专家的经验通过知识表示方法写成一条

一条规则，系统依照规则推理模拟专家的思维方式。不过，在实际应用中，专家系统都没有取得太大的成功，其最主要的局限性体现在系统明显受到规则数量的限制，规则数量决定了系统对不同情况的适应程度，然而规则是有限的，问题发生时的状况是无限的，用有限的规则处理无限的可能，注定是苦海无涯。

早期在 AI 方面取得成功的项目，多数解决的是具有明确规则和条件的问题，比如西洋跳棋。1997 年 IBM 的“深蓝”计算机在国际象棋上战胜人类世界冠军卡斯帕罗夫就是这个方面最著名的例子。对于人类来说，下象棋当然是很有挑战的项目，但是相比真实世界的复杂程度而言，国际象棋其实只是一个简单问题。棋盘上只有 32 个棋子和 64 个可以落子的位置，走法规则是非常明确的，所有可能的局面组合是有限的，可以被穷举出来，利用计算机的计算能力辅助以启发式搜索等算法，在摩尔定律的作用下，击败人类只是时间问题。在这类问题中，问题的表示通常都不是难题，一个普通程序员也可以在很短时间内完成一个象棋程序。然而许多真实世界的问题却并不都是那么容易能用计算机语言表达清楚的，比如图像识别和语音识别，这些问题有着比国际象棋大得多的问题域，即使对于人类来说，也有很多不能确定、无法选择的时刻，所以用规则来描述问题是不现实的。

因此，“演绎法”的规则推理暂时行不通，“归纳法”就成为了唯一的出路。基于概率统计的机器学习（**machine learning**）逐渐成为人工智能的主流方法。与专家系统不同，机器学习不会在系统中输入任何规则，而是直接在大量真实世界产生的数据中挑选最具有代表性的样本（**samples**）交给算法处理，让算法自动在数据中寻找和学习特定的规律，而这些由数据得来的规律，就是我们本来需要输入的规则。这种从数据中学习规律的过程也叫做模式识别（**pattern recognition**）。

机器学习的基本思路是假设样本数据与真实世界的概率分布相同，这样就可以认为算法从样本数据中归纳所得的规律在一般情况下同样适用。朴素贝叶斯（**naive bayes**）和逻辑回归（**logistic regression**）等算法都是机器学习的经典方法，并且都在实际应用中取得了很好的效果。

机器学习一般分为监督学习和无监督学习两种。监督学习要求每条样本数据都有对应的标签（**label**），样本数据作为输入，标签作为目标输出，学习的目标是求出输入与输出之间的关系函数 $y = f(x)$ ，使得针对每个输入样本 x ，都得到期望的输出结

果 y 。朴素贝叶斯、逻辑回归和神经网络等都属于监督学习的方法。而对于无监督学习来说,样本数据并没有标签,学习的目标是为了探索样本数据之间是否有隐含的不易被发现的关系,统计样本的分布情况,典型的算法有 K-means 等各种聚类算法。

监督学习主要解决两类核心问题,即回归 (regression) 和分类 (classification)。回归和分类的区别在于强调一个是连续的,一个是离散的。回归的输出可以是任意实数,而分类的目标输出为离散的类别编号,或是布尔类型的二值判断。分类问题可以利用概率模型以回归方式来求解,即认为样本所属的真实类别概率为 100%,样本属于其他类别的概率为 0%。通过将离散的类别编号转化为连续的概率,利用回归方法学习和预测样本属于每个类别的概率,概率最大的类别就是分类结果。

对于朴素贝叶斯和逻辑回归等简单的机器学习算法来说,本质上是要计算样本输入与目标输出之间的相关性。相关性固然是非常重要的,但在处理真实世界问题的时候,判断相关性其实是人们确定了所有影响因素之后的一个后续问题,各个影响因素的表示 (representation) 会严重影响对于相关性的判断。比如要辨别一段语音中演讲者是男人、女人还是小孩,简单机器学习算法的判断依据很可能主要来自音量而不是音色和音调。再比如,假设我们想用逻辑回归判断明年北京房价是否继续上涨,如果选定人口净流入数和地铁修建的里程数作为输入的话,会得到人口流入越多房价越高的结论,那么如果明年人口净流入减少,则系统会预测房价下跌。但真实情况是房价与货币增发量相关性更大,如果明年货币继续超发,最终房价还是会上涨,所以预测就可能会出现比较大的误差。

人口流入数、货币增发量这些对于预测房价有影响的因素,是我们从众多维度中提取的特征 (feature)。正如在前文例子中描述的那样,设计合适的特征表示在机器学习中是一项极其重要却又非常困难的工作。一方面,特征选取会直接影响预测的稳定性,要得到准确的预测结果就必须选中相关度最高的特征。在使用逻辑回归、朴素贝叶斯等简单机器学习算法的时候,由于特征选取问题而导致模型失效的情况比比皆是。但另一方面,如何设计特征又需要加入许多人类的先验经验才能完成,需要运用人类的智慧和经验分析各种因素所带来的直接或间接的影响。所以通常的做法是首先列举出各种可能的特征,然后通过交叉组合的方式进行穷举验证。在很长一段时间,特征工程 (feature engineering) 都是机器学习的重中之重,是每个研究员的必修技能。特征工程对于传统机器学习算法来说是如此重要,但同时也是机器学习应用的最

大束缚,不仅费时费力,还需要由人类提供大量的先验经验以弥补对数据本身挖掘不足的缺陷。若要拓展机器学习的适用范围,必须要降低学习算法对特征工程的依赖性。

深度学习 (Deep Learning, 也曾被叫作 Feature Learning) 是在人工神经网络 (Artificial neural network, ANN) 基础上发展而来的一种表示学习 (Representation Learning) 方法,也是一种机器学习方法,而且是人工智能领域最具发展前景的一个分支,如图 1-1 所示。其主要模型是各种深度神经网络 (Deep Neural Network)。表示学习是近年来机器学习领域发展最迅猛、最受学术界追捧的方向。所谓表示学习,就是要让算法在少量人为先验经验的情况下,能够自动从数据中抽取出合适的特征,完成原本需要通过特征工程才能得到的结果。在表示学习范畴中,深度学习是通过多层非线性变换的组合方式,得到更抽象也更有效的特征表示。原先使用机器学习解决主要工作包括需要大量人工处理的特征工程加上一个可被训练的分类器,而到了深度学习的时代,特征工程已经被各种可训练的特征提取器所取代,在应用效率上有了显著提高。更重要的是,人工智能目标就是让机器有能力理解我们所在的世界,只有当它能学会如何感知和辨别数据背后的各种隐含因素的时候才能达到这个目标。

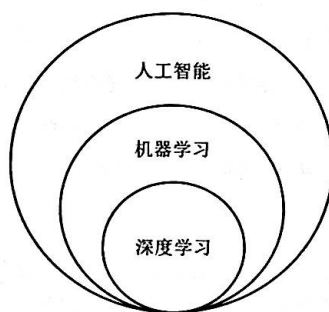


图 1-1 深度学习是一种机器学习方法,是人工智能领域最具前景的分支

真实世界的问题之所以困难,很大程度上是因为我们能够观察到的数据都是种种因素叠加而成的。比如同一朵花在白天和夜晚观察的颜色是不同的,同一辆车在不同角度观察的形状也不一样。表示学习最核心的诉求就是要发现真正重要的特征,舍弃那些并不影响判断的因素。要做到这种程度当然不是一件容易的事,那么深度学习又是如何做到呢?

深度学习是通过构建一个多层的表示学习结构,使用一系列非线性变换操作把从

原始数据中提取的简单的特征进行组合,从而得到更高层、更抽象的表示。在图像识别的场景中,图像在计算机中最基本的表示是一组像素值集合,从像素到物体的映射关系需要经过一个很长的过程,从像素组成细小的边,由边组成基础的纹理基元,纹理基元组合而成图形,图形构成物体的各种组成部分,最后组成物体的整体。同样,对于文本的理解也符合类似的过程,先认识各个字母,再由字母组成单词,单词组成词组,词组组成句子,句子组成段落,段落构成完整的故事。这个过程对于人类来说眨眼之间就已经完成了,但是对于计算机来说是非常复杂的,很难简单地一步求得这种映射关系。因此,深度学习模型的结构设计遵循了这种思路,具体做法是将一系列相对简单的非线性映射操作构建成一个多层网络,每一层(layer)都完成一次特征变换。以人脸识别为例,网络以像素表示的图像作为输入,在低级层次中主要学习到代表图像边缘的特征,可能是连续几个像素所组成的某个方向上的线段。中级层次会学习到由边缘线段所组成的局部图案,这些图案实际上是构成目标物体的各种部件,比如眼睛、鼻子、耳朵。在最后的高级层次中,以各种局部部件作为基本单元就可以组合出人脸的抽象表示,比如包括人脸上有会有一个鼻子两只眼睛、眼睛的相对位置在鼻子的两侧,等等。而符合这种抽象表示的图像,就可以被判定为人脸图片。如此多层学习结构中的中间特征如图 1-2 所示。

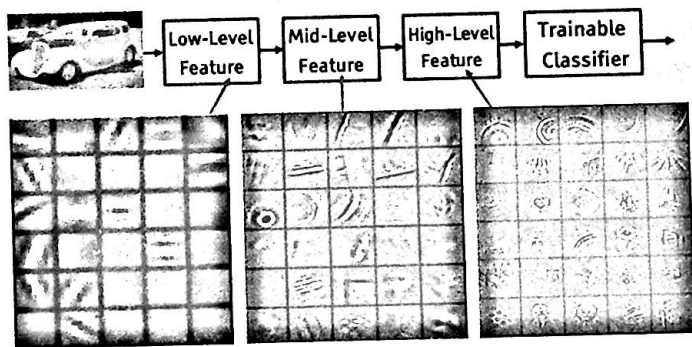


图 1-2 多层学习结构中的中间特征

那什么样的结构才算是有“深度”？其实包含三个以上隐层(hidden layer)的神经网络就可以说是一种深度学习模型,但由于在真实场景使用时参数过多、计算量过大,存在梯度消失和梯度爆炸的问题,无法做到稳定收敛,所以一般不会使用。一个网络的深度,可以以网络中串联的计算的层数,或者是非线性变换次数,甚至更加抽

象一些，以不同的计算概念来评估。关键在于，深度学习比传统机器学习模型多了多级特征提取的结构，能够进行表示学习，这样才算是有深度。从这个意义上说，只有两个隐层的神经网络就不能算是有深度的结构，因为它并没有能力分级提取特征。

了解人工神经网络知识的读者可能会有一个疑问，具有一个隐层的人工神经网络已经非常强大了，理论上来说，只要神经元足够多，构成一个很“宽”的网络，它可以拟合任意的函数，那为什么还要更“深”？原因在于：在神经元数量相同的情况下，深层网络结构具有更大的容量，分层组合带来的是指数级的表达空间，能够组合成更多不同类型的子结构，这样可以更容易地学习和表示各种特征。并且，隐层增加则意味着由激活函数（activation function）带来的非线性变换的嵌套层数更多，就能构造更复杂的映射关系。

现如今，深度学习已经在多种应用上取得了突破性进展。卷积神经网络（**Convolutional Neural Networks, CNNs**）是这一波深度学习浪潮的引领者。2012 年 AlexNet 在 ILSVRC 图像识别竞赛中所带来的惊人表现，让学术界看到了深度学习所蕴含的巨大潜力。在随后的几年中，随着 GoogLeNet、VGGNets、ResNets 等模型的提出，CNNs 的识别准确率持续提高，让计算机拥有了超越人类的图像识别能力。关于 CNNs 的应用将在本书第 4 章中详细介绍。同时，深度学习在自然语言处理（**Natural Language Processing, NLP**）方面同样取得了巨大的成功，不但有 Siri 这样可以与人类正常交流的对话机器人，甚至能够写诗、作曲。本书第 5 章会介绍循环神经网络（**Recurrent Neural Networks, RNNs**）在处理语音或者文字等问题中的应用。

总结来说，深度学习是一种机器学习方法，同时也是目前最有希望具有处理复杂的真实世界问题的能力的人工智能方法。深度学习的分层结构能够从简单概念的组合中学习到高级抽象的特征表示，让计算机真正具有理解世界的能力。不同类型人工智能系统的流程图如图 1-3 所示。

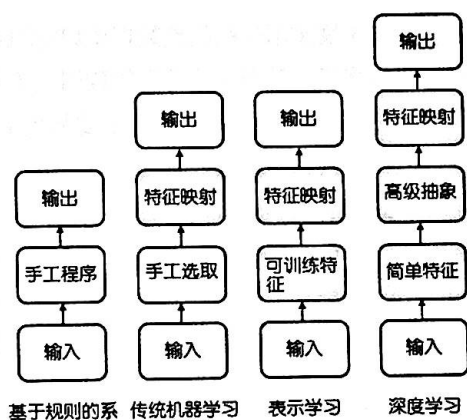


图 1-3 不同类型人工智能系统的流程图。灰色框表示具有从数据中学习的能力

1.2 深度学习的趋势

20 世纪 90 年代，在卷积神经网络应用于文字识别之后，深度学习曾经经历一段时间的沉寂，而 2012 年在图像识别领域的爆发式突破，让人们意识到其巨大的潜力和广阔的应用前景。现如今，以 Google、Facebook、Nvidia 等为首的科技公司全都重兵押注深度学习，而国内的百度、阿里巴巴、腾讯也紧随其后，让人深深感觉到这是一个朝气蓬勃的领域，同时也是一个硝烟弥漫的战场。

很多人会好奇，人工神经网络几十年前就已经存在了，为什么直到今天才认为深度学习是一项突破性的技术？事实上，在 20 世纪 90 年代深度学习就已经有成功的商业应用，但是在那时，人们更多会认为这是一门只有少数几个专家才能掌握的艺术而不是技术。深度学习的成功，依赖于四项基本要素：

- 海量的训练数据；
- 非常灵活的模型；
- 足够的运算能力；
- 足够对抗维度灾难（curse of dimensionality）的先验经验。

机器学习是要从数据中学习知识，正所谓“巧妇难为无米之炊”，即使再厉害的算法，也需要优质的数据集支持。所以在人工智能几十年的发展过程中，数据集的种

类和规模都在逐渐增长。图 1-4 展示了一些经典数据集规模的对比。从中可以看到, 样本的规模几乎是在以指数级增长, 其中一些著名的数据集 (如 ImageNet) 的样本数量已经达到了千万级别。原先, 机器学习的任务是, 要从少量数据中学习特定的规律, 然后泛化到更一般的场景中, 并且要防止各种欠拟合和过拟合。但随着大数据时代的到来, 数据集本身就已经包含了各种可能出现的情况, 从数据集中学习的规则可以直接应用在现实场景中。以 2016 年的趋势来看, 对于有监督的深度学习算法来说, 只要每个类别有 5000 个样本, 机器就能达到令人满意的表现, 而若有 1000 万以上的样本, 机器就能达到甚至超越人类的水平。在互联网已经深入人们生活每一个角落的今天, 数据更是会越来越多, 各种“独角兽”公司每天产生的数据量是 PB 级别的。当收集数据已经不再是难题, 如何更有效地利用数据就成为了新的挑战。

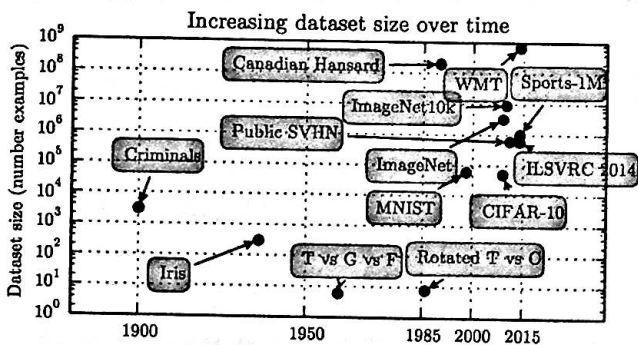


图 1-4 在人工智能学科刚被确立的时候, 研究的数据仅在几百条的规模。到了 20 世纪 90 年代, 著名的 MNIST 手写数字数据集有 60000 多张 28×28 分辨率的黑白图片。而到了近些年, ImageNet 数据集超过了 100 万张大尺寸彩色照片

深度学习涉及大量的科学计算。计算机硬件的性能提升也为深度学习发展推广提供了基本必要条件。著名 CPU 厂商英特尔 (Intel) 的创始人之一戈登·摩尔 (Gordon Moore) 曾提出摩尔定律, 其内容为: 当价格不变时, 集成电路上可容纳的元器件的数目, 约每隔 18~24 个月便会增加一倍, 性能也将提升一倍。摩尔定律概括了 20 世纪末一段时间 CPU 的发展速度。到 2017 年, 家用电脑 CPU 计算频率已经接近当前技术的上限。因此, 除了靠提升单个 CPU 的计算频率来提升计算速度外, 人们还想到使用多核或多 CPU 集成的方式来提升计算性能。CPU 能够快速处理多种类型的计算, 但并不是每种类型的计算耗时都相同的 (例如双精度浮点数乘、除法和条件判

断语句)。因此多 CPU 集成, 同样受到频率同步和稳定性的限制。我们可以看到, 市面上好几万元的品牌机服务器的 CPU 核心往往是 16~64 个, 而主频一般都在 3.0GHz 以下。而几千元的 PC 主频虽然可以达到 3.0GHz, 而 CPU 核心一般是 2~8 个。

考虑到深度学习的特殊计算需求和 CPU 综合计算性能方面的这些限制, 人们考虑使用 GPU 来解决科学计算问题。20 世纪 90 年代, PC 电脑发展初期 GPU 的主要功能是用于显示交互界面。其画面的显示功能涉及坐标点变换、栅格化、平面渲染等大量的浮点数加、减、乘、除等计算操作。对比来说, CPU 是面向通用计算的产品, 而 GPU 则是天生面向大规模浮点数并行计算的。与 CPU 相比, GPU 的内置计算核心虽然计算频率相对较低, 但内置计算核心数量更多。以著名 GPU 厂商英伟达 (Nvidia) 为例, 近几年推出的 Tesla、Titan X 系列显卡处理器数量都以上百、上千计, 而计算主频在 1GHz 左右。在 GPU 的帮助下, 神经网络的训练效率大幅提高。与同等价位 CPU 相比, 速度提升了约几十倍, 分析大型的深度网络结构的效率因此得到了提升。英伟达公司 Tesla 系列显卡与 CPU 芯片的计算能力比较如图 1-5 所示。

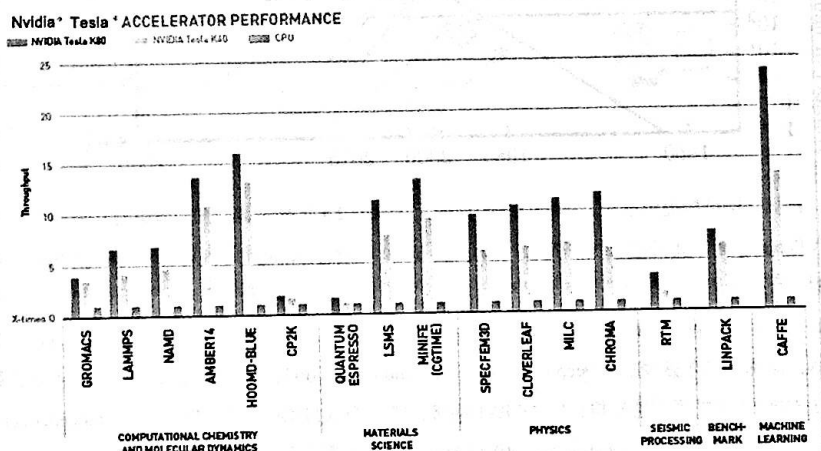


图 1-5 Tesla 系列显卡与 CPU 芯片的算力比较。在科学计算方面 GPU 显卡是 CPU 速度的数倍

随着计算能力的提升和更大规模数据集的出现, 神经网络模型不但结构越来越复杂, 而且规模也在不断增大。尤其自从隐层结构提出以来, 神经网络的神经元数量每 2.4 年翻一番, 如图 1-6 所示。对于深度学习来说, 人类赋予其最重要的一个先验经验就是: 分层组合的表示方式更能准确地描述我们所在世界的规则。因此, 深度学习

2

TensorFlow 系统介绍

随着各种图像识别、语音识别的纪录被不断刷新，深度学习也被证明是一个极具潜力的技术方向，越来越多的研究学者涌入这一领域尝试新思路、新方法、新的应用方向。在这样的背景之下，构建高效、可靠、可扩展的基础工具，能为这一领域的发展起到极大的推动作用。

TensorFlow 是 Google 推出的一套深度学习系统，使用其灵活易用的前端语言可以轻松地构建各种复杂的算法模型，后端高效的执行系统与分布式架构保证了在模型训练和执行方面的高性能。综合来看，TensorFlow 是目前业界最优秀的深度学习系统之一。

在本章中，从系统需求、基础概念、系统架构和源代码几个方面对 TensorFlow 系统进行了介绍。正所谓“磨刀不误砍柴工”，在开始实战之前先了解一下 TensorFlow 内部的设计原理和运行机制，相信可以帮助读者更好地运用这一强大的工具。

2.1 TensorFlow 诞生的动机

Google 可以说是人工智能方面工业界的领军者之一。早在 2011 年，就由 Google 传奇人物 Jeff Dean、Google 研究学者 Greg Corrado，携手斯坦福大学教授 Andrew Ng

共同创建了 Google Brain(谷歌大脑)计划,旨在探索深度学习在实际场景中的应用。多位专家的联合努力,基于 Google 已有的云计算基础架构,设计开发了第一代机器学习系统 DistBelief。DistBelief 继承了 Google 所有分布式系统的长处,具有高性能、高扩展性等特点。在 Google 内部,有包括搜索、地图、翻译等超过 50 个团队使用了基于 DistBelief 构建的深度神经网络算法模块。

深度学习如今发展速度越来越快,大量新颖的研究成果不断涌现,能够应用深度学习技术的场景也越来越多。作为学术研究和工程产品的基础,深度学习框架系统在其中起到了巨大的促进作用。基于在多个项目上研究应用的经验,Google Brain 团队总结出了对于深度学习系统的以下几点核心需求。

要具有灵活的表达能力,能够快速实现各种算法

随着越来越多的专家学者加入了深度学习领域,几乎每天都有各种疯狂的新想法、新模型涌现出来。快速实现算法、快速验证结果,成为了主要的需求之一。在曾经的 MATLAB 时代,算法的实现至少需要有经验的专家花费几天的时间,即使在 Caffe 等框架出现之后,新模型(如 RNN 等)的扩展仍然是一件困难的事情。因此,对神经网络进行合理的抽象封装,并设计出灵活的编程接口,支持各种模型的组合与扩展,是深度学习系统的首要目标。

高执行性能,具备分布式扩展性

由于机器学习算法执行的计算量通常都比较大,所以程序实现必须经过充分的优化,减少不必要的开销,以降低执行时间。因此,对于深度学习框架系统来说,执行性能是非常重要的基础评价指标。不仅如此,在服务器集群大行其道的今天,若仅依靠单机运算是远远不够的,而利用分布式技术进行并行计算可以大大提高计算吞吐量,进一步提升效率,所以支持分布式扩展对于深度学习系统来说也是非常重要的。

跨平台可移植性

跨平台已经不是什么新鲜的概念了,桌面应用会跨 Windows 和 MacOS 系统,手机 APP 要同时支持 iOS 和 Android 平台。机器学习算法要想应用于生产生活场景,也需要适应多种平台载体。随着深度学习技术日趋成熟,应用会越来越贴近人们的生活,将模型运行在手机,甚至是手表等随身设备上,应该会成为主要的趋势。因此,

由深度学习系统提供跨平台能力,既能让模型在 GPU 服务器上高效训练,又能快速部署在各种生产环境中,可以为实用带来极大的便利。

实验可复现性

如今是讲究开放和合作的时代,交流与研究同等重要。大部分机器学习算法都是基于概率论发展起来的科学,因此很多的实验都具有一定程度的随机性,这就为实验的精准复现带来了难度。深度学习框架系统可以从底层提供解决这一问题的工具,让实验不再是一次性的结果,而是可重现、可共享的资料。这对促进研究的发展有着极其重要的作用。

支持快速产品化,模型可随时部署

随着深度学习的快速发展,一些模型和技术已经具备了处理真实世界问题的能力,例如识别各种文字、识别语音、识别图像,等等。对于以往的一些技术而言,研究和应用之间往往存在较大的间隔,从学术界到工业界有一个转化的过程。但深度学习不同,在问题定义之初可能就已经是一个实际的具体问题,只有很少的抽象成分,所以往往模型只要一训练好,就可以立刻投入到生产系统中使用。因此,深度学习系统应该支持模型在训练和应用两方面灵活切换,随时将研究成果集成到产品中。

虽然第一代深度学习系统 DistBelief 已经拥有很好的扩展性,但在灵活性方面的不足^[3],使得 Google Brain 团队有动力开发第二代系统,即 TensorFlow。它对于机器学习应用来说就如同 Linux、iOS 等操作系统一样,一方面为用户构建上层应用提供接口,让用户以此为平台,开发出各种各样的应用产品,另一方面管理和控制底层的计算机硬件和软件资源,以提高资源利用率,降低硬件差异等问题所带来的研发成本。

2.2 TensorFlow 系统简介

深度学习的研究始终都是为了解决现实世界中存在的问题,所以在学术界不停探索新模型、新方法的同时,工业界也不断推出各种工具框架,促进研究和应用的发展。Google 经过长期的研究和尝试,在实践的基础上推出了目前最优秀深度学习框架系统之一——TensorFlow,其项目 logo 如图 2-1 所示。

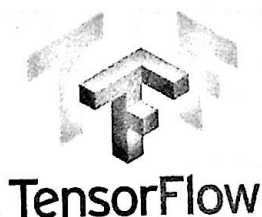


图 2-1 TensorFlow 项目 logo

在 TensorFlow 的官方介绍中给出的定义是：TensorFlow 是一个基于数据流图（Data flow Graph）的数值计算开源软件库，其灵活的架构设计可以让用户以单机或分布式的方式将计算部署在台式机、服务器，甚至是手机上。通过这个描述可以看出，首先，TensorFlow 面向的是数值计算，虽然起初是作为深度神经网络方向研究的工具，但它在通用计算方面的应用范围也不容小觑。其次，数据流图（或称为计算图）是 TensorFlow 中对于计算过程的抽象表示。数据流图是有向图，图中的点表示各种数学计算操作，边表示操作与操作之间传递的高维数组数据，称为 tensor。最后，支持各种设备的灵活部署是 TensorFlow 最大的优势，大至分布式服务器集群，小到手机等移动设备，都能运行同样的机器学习算法模型。

TensorFlow 作为一个机器学习框架系统，主要解决了机器学习方面的几个核心问题。

首先，使用 TensorFlow 可以用非常简洁的语言实现各种复杂的算法模型，将研究从极为消耗精力的编码和调试工作中解放出来。相比早期研究所用的 MATLAB，使用 TensorFlow 实现的代码行数只是之前的一半甚至更少。代码少就意味着更不容易出错，也更容易排错（debug），仅这一项就可以在实际开发中可以节省大量的时间。

其次，TensorFlow 的内核执行系统使用 C++ 编写，因此保持了非常高的执行效率。在同类系统的横向评测中，TensorFlow 始终保持在领先地位，并且首先开源了完整的分布式的加速方案，实验中 50 个设备的并行化方案能够达到 30 倍以上的加速效果，在执行效率上有巨大的提高。

再次，TensorFlow 优秀的分层架构设计使得模型可以非常方便地运行在异构设备环境上。TensorFlow 采用前端编程语言与后端执行引擎分离、执行引擎又与面向硬件

的计算实现分离的架构，充分实践了“高内聚、低耦合”的设计思想。可以非常方便在底层增加算子，甚至新增对特殊硬件设备的支持，而在上层则可以不用担心硬件运行的特殊机理，专心在算法和应用上取得突破。

最后，TensorFlow 不仅包含 TensorBoard 等优秀的配套辅助工具，第三方社区也贡献了诸如 TFLearn 等众多好用的辅助项目，不仅使代码更简洁，也让数据处理工作变得轻松容易。

2.3 TensorFlow 基础概念

2.3.1 计算图

TensorFlow 是一个基于计算图 (Computational Graph, 也叫作数据流图 Data flow Graph) 的数值计算系统。计算图是一个有向图，图中的节点代表数学计算操作的算子 (operations, 简称为 op)，节点之间连接的边代表参与计算的高维数组数据，叫做 **tensor**。计算图的执行可以看作数据 **tensor** 按照图的拓扑顺序，从输入节点逐步流过所有中间节点，最后流到输出节点的过程，TensorFlow 的名字由此而来。基于 TensorFlow 的计算图如图 2-2 所示。

tensor (张量) 这一术语起源于力学，在物理和数学中都有重要的作用。在 TensorFlow 系统中，**tensor** 代表多维数组 (multi dimensional data array)，对应神经网络计算中的高维矩阵。**tensor** 可以有任意维度，每个维度也可以有任意长度。特别来说，一维 **tensor** 就是向量，是普通的一维数组；二维 **tensor** 是矩阵。TensorFlow 中一般使用 4 维 **tensor** 表示一个 mini-batch 的图片，四个维度分别是批大小、像素行数、像素列数、通道数，即 [batch, height, width, channels]。**tensor** 中的元素可以是任意内置类型，常用的有：int32、int64、float32、float64，等等。对于神经网络来说，单精度 float32 应该是最常用的数据类型。

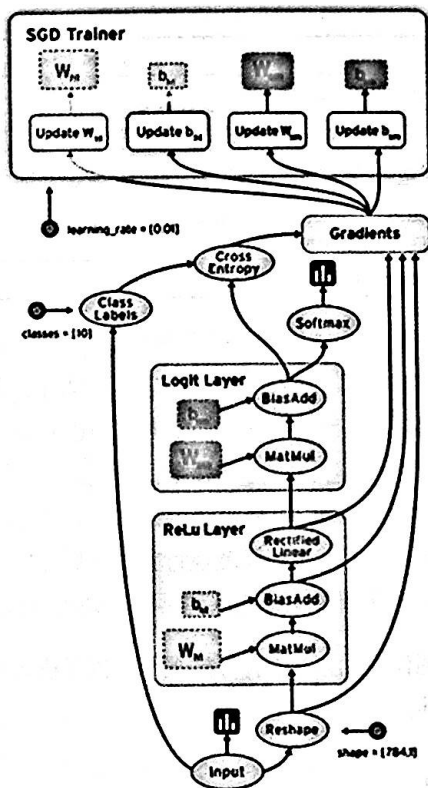


图 2-2 基于 TensorFlow 的计算图示意

算子 `op` 是参与计算的基本单位，每个算子都对应一种数学运算，比如：矩阵加法、矩阵乘法等。算子接收 0 个或多个 `tensor` 作为输入，进行一定的计算，输出 0 个或多个 `tensor` 的结果。在实现中，一些算子可以带有特定的属性，比如说矩阵运算算子带有数据类型 `dtype` 属性，设置数据类型可以实现计算的多态，既可以完成 `int32` 类型数据的运算又可以进行 `float64` 类型的运算。TensorFlow 内置了多种算子，表 2-1 列出了部分类别的算子示例。

表 2.1 TensorFlow 集成的算子示例

类 别	算 子 示 例
数值计算操作	Add, Sub, Mul, Div, Less, Equal, ...
数组操作	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...

续表

类 别	算 子 示 例
矩阵计算操作	MatMul, MatrixInverse, MatrixDeterminant, ...
状态操作	Variable, Assign, AssignAdd, ...
神经网络相关操作	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
存档操作	Save, Restore
队列和同步操作	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
控制流操作	Merge, Switch, Enter, Leave, NextIteration

与其他一些使用数据流作为计算抽象的系统（如 Spark、Tez 等）有所不同的是，TensorFlow 的计算图对有向图概念进行了一些扩展，加入了图的状态表示。对于神经网络等算法来说，有向图可以表示模型的结构，但图中边的权值参数并不是固定不变的，而是通过训练过程不断调整更新的。所谓图的状态，就是图中所有参数的组合。在 TensorFlow 中，以变量（Variable）来存储参数值。变量可以与 tensor 一样参与各种运算，区别在于 tensor 的值在每次计算图执行完成之后立即被丢弃，而变量的值在通过反向传播计算更新后会保留下来，代入到下一轮训练迭代。

TensorFlow 以计算图作为抽象，为计算的表示提供了极大的空间和自由度，保证了各种算法实现的灵活性。

2.3.2 Session 会话

Session 是驱动 TensorFlow 系统执行计算交互的入口。Session 负责完成多计算设备或集群分布式的节点布置和数据传输节点的添加，并负责将子图分配给相应的执行器单元来运行。

从使用角度说，典型的用法是客户端通过 CreateSession 接口与 master 建立连接，并在初始会话的过程中传入计算图。对于 Python 接口，计算图可以在 Session 创建之前构造完成，并在 tf.Session 对象初始化时载入到后端执行引擎。Session 还提供了 Extend 的接口，可以在会话中修改计算图。

触发计算图的执行通过 Run 接口，也就是 Python 的 tf.Session.run()方法。通过这个接口，可以将数据代入模型，执行计算，并得到执行结果。训练过程由客户端的循环来控制，一般情况下，都会在一个会话中通过 Run 接口执行成千上万次的计算。

Session 管理了运行时的一系列资源的申请和分配，所以在计算完成后，必须要关闭 Session 以释放资源。

TensorFlow 的 Python 接口典型示例如下：

```
import tensorflow as tf
b = tf.Variable(tf.zeros([100]))          # 100 维向量，初始化为全 0
W = tf.Variable(tf.random([784,100],-1,1)) # 784x100 矩阵，随机初
始化的变量
x = tf.placeholder(name="x")              # 输入占位符
relu = tf.nn.relu(tf.matmul(W, x) + b)    # Relu (Wx+b)
C = [...]                                  # 代价函数
s = tf.Session()
for step in xrange(0, 10):
    input = ...construct 100-D input array ...# 100 维的输入数据
    result = s.run(C, feed_dict={x: input})  # 代入 x=input,
                                              # 获取代价函数的计算结果
print step, result
```

2.4 系统架构

TensorFlow 是拥有“client → master → worker”架构的分布式系统。在一般的执行流程中，客户端通过会话 tf.Session 接口与 master 进行通信，并向 master 提交触发执行的请求，master 将执行任务分配到一个或多个 worker 进程上，执行的结果通过 master 返回给客户端。其中，worker 是最终负责执行计算的角色，每一个 worker 进程都会管理和使用计算机上的计算硬件设备资源，包括一块或多块 CPU 和 GPU，来处理计算子图（subgraph）的运算过程。

计算设备（device）是 TensorFlow 系统中对计算资源定义的基础单位，由 worker 进程管理和使用。TensorFlow 官方支持 CPU 和 GPU，即每个 CPU 核或每块 GPU 都是一个计算设备。每个抽象的计算设备要负责该硬件芯片上的内存分配与释放，并且执行上层应用所分配的计算。在系统中，每个计算设备都被分配了标识名称，该名称由三部分组成：worker 所属的任务名、计算设备的类型、硬件在 worker 中的编号。例如，“/job:localhost/device:cpu:0”代表单机 worker 中第一个 CPU 芯片设备，“/job:worker/task:17/device:gpu:3”代表分布式环境下第 17 个 task 所在机器的第三块 GPU 显卡。有了标识名称，上层应用可以显式地手动指定计算设备。虽然目前官方

仅支持 CPU 和 GPU 两类设备，若想添加新的硬件设备也并不困难，TensorFlow 的实现中将算子的定义和面向硬件的代码实现进行了分离，可以单独开发所有算子在新设备上执行代码，并通过注册机制注入系统，即可实现计算设备的扩展。

TensorFlow 是一个可以灵活横向扩展的架构。在单机模式下，一般 client、master 和 worker 都在同一个进程之中启动，worker 进程会管理本机上所有的计算设备。而在分布式环境下，client、master 和 worker 之间会通过远程调用的方式连接在一起，每个 worker 各自独立管理执行机上的计算设备。TensorFlow 的单机与分布式架构如图 2-3 所示。

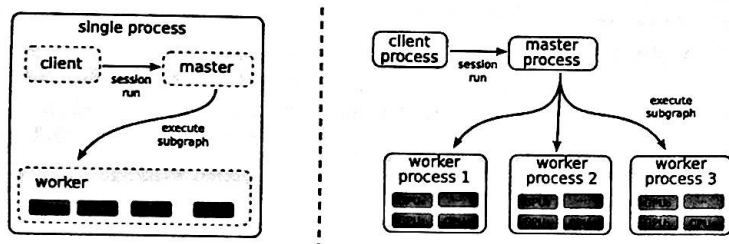


图 2-3 TensorFlow 的单机与分布式架构

从计算图的执行角度来说，当只有一个执行计算的硬件设备时，计算图会按照节点的依赖关系顺序执行。

在单机多设备模式下，为了提高计算性能，会将计算图分解成若干个子图，每个子图被分配到不同的设备上执行。此时主要会面临两个问题，一个是要决定每一个计算节点安置在哪个硬件设备上执行，二是在不同设备之间构建数据交互的通道。TensorFlow 系统要做的是要将计算图中的每一个节点与硬件的计算设备做一个映射，将算子的执行安置在合适的计算硬件设备上。这个安置过程由布置算法（placement algorithm）决定。基础布置算法会模拟计算图的执行，针对每一个算子节点，系统利用代价模型推算该操作在所有可用设备上的预计耗时，使用贪心算法选出耗时最小的设备。在计算代价时，会考虑设备之间数据复制的问题，数据在设备的内存间复制传输往往会有比较大的时间消耗。

当计算图的算子被布置到不同设备上时，系统会自动在跨设备的操作之间加入数据传输算子，如图 2-4 所示，系统会成对添加发送（send）和接收（recv）算子。此

时, 每个设备实际只负责子图的执行, 子图执行所得的结果 **tensor** 会通过发送和接收数据的算子在设备之间复制传输。添加通信节点的工作由系统自动完成, 对于上层应用来说, 编写的程序与单机版本并无二致, 可以说极大简化了分布式的开发工作。更加值得一提的是, 在程序实现中, 数据传输是根据需要在发送和接收节点之间自动触发的, 完全是一种去中心化的组织方式, 无需 **master** 在其中参与调度, 这可以让系统的横向扩展性得到极大提高。

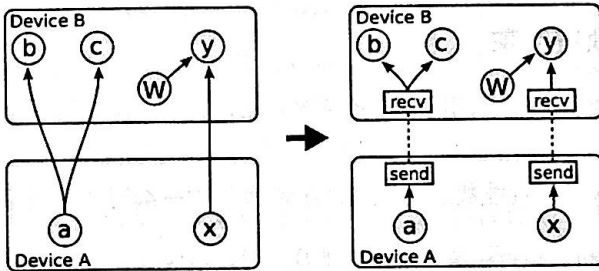


图 2-4 跨设备布置算子时, TensorFlow 会自动添加发送和接收数据的节点

在集群分布式环境中, 执行方式与单机多设备模式比较类似, 区别仅为节点通信是跨机器的网络通信。对于集群来说, 最大的问题是容错机制。TensorFlow 有两种检测错误的机制, 一种是基于发送和接收算子之间传输的错误信息, 另一种是 **master** 进程的轮询检查。当发现错误后, 系统会终止当前迭代, 从存档状态重新运行。

TensorFlow 的分布式架构, 既保证了单机环境下的易用性, 又让用户在有需要时可以非常容易地扩展计算规模, 利用集群优势提高训练效率, 充分体现了 Google 工程师们杰出的工程架构能力。

2.5 源码结构

TensorFlow 的系统实现采用了分层的结构, 核心是由 C++ 语言实现的后端执行系统, 前端则提供了基于多种语言的编程接口。在保证执行性能的同时, 又降低了学习门槛和应用集成的代价。

TensorFlow 系统架构的示意图如 2-5 所示。

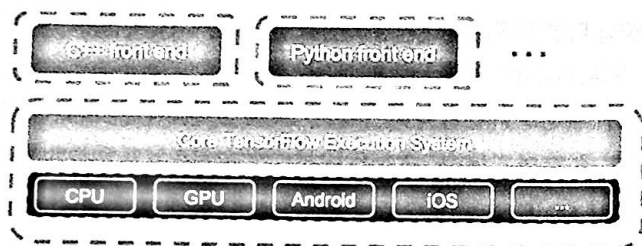


图 2-5 TensorFlow 系统架构示意图

2.5.1 后端执行引擎

TensorFlow 的后端执行引擎是整个系统的核心，对应源代码中的 `tensorflow/core` 目录，在 TensorFlow 系统中，计算执行部分的逻辑都在此部分实现。执行系统的实现借鉴了 NumPy 等高性能数值计算库的方案，使用 C++ 编写，以充分保证执行性能。

从代码来分析，执行引擎主要由以下几个模块组成。

1. 系统框架（对应 `core/framework` 目录）

框架主要实现 TensorFlow 系统的基本运行方式。其中定义了计算图、`tensor`、算子等数据结构，还包括数据类型、资源管理器、消息通信机制等基础设施。

在算子的实现中，TensorFlow 使用了注册机制。框架并不关心系统都实现了哪些算子，而是在运行时按注册名称查找算子的实现。这种方式对工程扩展是非常有益的，算子的实现内聚性高，不用担心在扩展新的计算类型时对系统造成意想不到的破坏。

2. 计算图（对应 `core/graph` 目录）

计算图模块主要实现了计算图模型的数据结构，包括节点 `Node`、边 `Edge` 和图 `Graph` 等类的定义，以及图的遍历算法和在布置节点时所需的代价模型。

3. 算子声明与内核实现（对应 `core/ops` 和 `core/kernels` 目录）

TensorFlow 作为一个数值计算系统，包含了种类丰富的数学计算算子。算子的声明和实现是分离的。注册声明使用 C++ 的宏 `REGISTER_OP`，其中定义了算子的名称、输入输出、算子相关参数以及说明文档。内置算子的注册声明在 `ops` 目录中。

计算操作在硬件上的具体代码实现被称为内核（`kernel`）。这个实现与计算平台相

关,也就是说,同样的计算操作在 CPU 和 GPU 上的运行代码是不同的。对于 CPU 计算而言,数值计算会基于 Eigen 库。Eigen 库是一个 C++ 编写的模板库,支持整数、浮点数、复数,使用模板编程,可以为特殊的数据结构提供矩阵操作。而对于 GPU,计算会基于 Nvidia 的 CUDA 库来完成。CUDA 是计算统一设备架构(Compute Unified Device Architecture)的简称,它是一个并行编程平台,提供了基于 GPU 加速的编程 API。

4. 前后端交互接口(对应 core/protobuf 目录)

TensorFlow 前后端交互接口使用 protocol buffer 协议定义。在源码目录中,最主要的是 master_service.proto 和 worker_service.proto 两个文件,分别声明了 master 和 worker 服务的接口。这部分的定义符合了前文介绍的分布式架构。

除了服务 API 定义以外,其他 proto 文件中定义了各种必要的数据结构,比如有 master 和 worker 服务接口的请求数据结构及响应数据结构,有执行时设置的 GPUOptions 和 GraphOptions 等配置的参数结构,还有用于存档模型的 MetaGraphDef 等。

5. 运行时(对应 core/common_runtime 和 core/distributed_runtime 目录)

运行时模块负责管理程序执行时所需要的资源,其中包括对设备 device 的定义、内存分配 BFC(best-fit with coalescing)算法,以及计算图执行器和执行状态收集器。此外,运行时模块还包含了节点布置的 SimplePlacer 算法的实现。

在分布式运行时模块中,主要实现了 master 和 worker 两个服务,以及执行调度器。

6. 操作系统平台(对应 core/platform 目录)

该模块主要的作用是隔离不同的底层操作系统。由于在 Linux、Windows、Android、iOS,甚至 Google 内部的操作系统上,底层的接口各不相同,需要引用的库也不相同,所以先对底层操作进行封装,可以降低跨平台运行的开发成本。

7. TFRecord 数据的格式定义类 Example(对应 core/example 目录)

Example 类是在使用 TFRecord 格式数据时必须用到的数据格式声明方法。关于

TFRecord 格式, 将在第 3 章介绍。

2.5.2 前端语言接口

TensorFlow 目前已经支持了 C/C++、Python、Java、Go 等多种前端语言的接口。前端接口最主要的作用是利用宿主语言的优势对 client 进行封装, 使得系统更加易用。

以 Python 前端为例, 在计算图的构建方面, 使用了符号式编程 (symbolic programming) 的定义方式, 即在计算图的构建阶段, 命令的执行并不会真正进行数学运算, 而是仅完成计算节点的声明。这与一般的命令式编程, 即每执行一条语句就能得到相应的计算输出, 在使用上存在一定差别, 在实现一些复杂的代价函数时需要注意转换思路。

TensorFlow 系统的核心是计算, 也提供了多达数百种的算子。对于算子的接入, 如果依靠人工逐个建立前后端 wrapper, 工作量会是非常巨大的, 而且针对不同语言也需要重复实现几乎相同的逻辑, 效率比较低。为此, TensorFlow 的代码中充分采用了代码生成技术, 对于 Python 语言版本, 前后端使用 swig 生成的代码相连接, Go 语言版本则是实现了一个代码生成器, 在编译期间生成所需要的代码。

2.6 小结

TensorFlow 自 2015 年 11 月开源以来一直受到业界热烈的追捧, 是 GitHub 上机器学习门类中最受关注的项目。截至 2016 年年底, 已经获得超过 4 万个 Star, 被 fork 了超过 2 万次。

TensorFlow 作为目前深度学习领域最优秀的项目之一, 在设计理念上, 充分吸收了 Google 在众多项目上的实践经验, 拥有合理的抽象表示; 在工程上, Google 优秀的工程师和开源社区共同保证了系统实现的高质量与极强的可扩展性。这都为深度学习领域的研究和应用实践铺平了道路。

相信在业界共同努力下, TensorFlow 这样一个重量级项目必将长时间持续发展下去。

2.7 参考资料

- [1] TensorFlow 官方网站: <https://tensorflow.org/>
- [2] TensorFlow 白皮书, <https://arxiv.org/abs/1603.04467>
- [3] TensorFlow 源代码: <https://github.com/tensorflow/tensorflow/>
- [4] TensorFlow playground: <http://playground.tensorflow.org/>
- [5] Jeff Dean & Oriol Vinyals. "Large Scale Distributed Systems for Training Neural Networks". NIPS 2015 Tutorial.
- [6] 谷歌大脑 Wikipedia 页面: https://en.wikipedia.org/wiki/Google_Brain
- [7] TensorFlow Wikipedia 页面: <https://en.wikipedia.org/wiki/TensorFlow>

3

Hello TensorFlow

TensorFlow 是一套深度学习开源软件库。从实现上来说，它的计算核心是基于高性能 C/C++ 实现的，而外部封装使用的则是易用的 Python 语言。这样的设计思路不仅保证了软件库函数的性能，还降低了使用的难度。简单来说，使用 TensorFlow 就是使用 Python 软件库编写深度学习程序。因此，在介绍 TensorFlow 使用的同时，本章还会概括一些关于 Python 程序的相关内容。

本章首先介绍了开发之前的环境准备工作，包括 TensorFlow 的安装注意事项和一些常用的辅助函数库。随后通过解决泰坦尼克号幸存者预测这样一个典型的分类问题，使用 TensorFlow 实现标准的神经网络分类器。通过这个例子介绍了 TensorFlow 提供的强大功能，并进一步概括了深度学习相关的优化技巧。

3.1 环境准备

在看完前面的各种介绍之后，相信各位读者一定都迫不及待地想要开始开发训练自己的模型了！但是还请稍安勿躁，第一步先来准备好开发运行环境。

在操作系统方面，TensorFlow 主要支持 UNIX 内核的操作系统，对于 Windows 系统的支持并不全面。这对于资深 Windows 用户来说并不是个好消息。虽然可以用

虚拟机或者 Docker 之类的虚拟化方法运行，但是且不说存在比较严重的性能损失，运行起来仍可能存在各种小问题。所以，Mac OS 或者 Linux 系统是安装 TensorFlow 的基础必要条件¹。

从开发语言方面讲，TensorFlow 的 Python 接口是功能最全且最简单的调用方式。而 C/C++ 接口虽然运行效率更高，但毕竟编写和调试的门槛略高。TensorFlow 对 Python 2 和 Python 3 都有非常全面的支持，所以对于 Python 的版本没有特别的要求，可以自由选择。本书主要以 Python 2.7 版本为主。

在硬件方面，作为需要大量计算的深度学习程序，非常需要有强大计算性能的底层硬件给予支持。GPU 显卡是目前执行高精度浮点型计算性能最强的设备，所以 GPU 服务器是深度学习研究中不可或缺的部分。理想情况下，可以在单机环境中用小规模数据完成代码调试，调试通过以后，在高性能的 GPU 服务器（或服务器集群）上使用大数据对模型进行训练。TensorFlow 目前支持 Nvidia 出品的多款显卡，只要支持 CUDA 7.0 以上驱动即可，可根据实际需求配备。

3.1.1 Mac OS 安装

在 Mac OS 系统下，TensorFlow 的安装过程非常简单方便（千万不要说在 Mac 的电脑上装了 Windows 系统）。Mac OS 是基于 UNIX 的操作系统，系统中已经基本包含了所有 TensorFlow 需要的依赖组件。

在跟着官方教程安装之前，建议使用 Homebrew 重新安装 Python。主要原因是，虽然 Mac OS 自带的 Python 在功能上同样完整，也可以正常使用，但部分系统路径与使用源码安装的版本略有差别，为了避免一些不必要的麻烦，推荐使用 brew 重新安装 Python 2.7²。另外，重新安装 Python 的一个好处是，使用 pip 安装 Python 库时不再需要 sudo 权限。

1 TensorFlow 从 0.12.0 版本开始增加了对 Windows 系统的支持，但还存在一些明显的限制，如不能使用 HDFS 存储、不能加载自定义算子、一些内置算子还未实现等。具体详情请参见 TensorFlow 的版本发布声明：<https://github.com/tensorflow/tensorflow/blob/master/RELEASE.md>。

2 Homebrew 推荐使用中科大镜像源，设置方法参见：<https://lug.ustc.edu.cn/wiki/mirrors/help/brew.git>。

```
$ brew install python
```

通过 Homebrew 安装的 Python 的位置默认为 `/usr/local/Cellar/python`。

TensorFlow 可以通过二进制包或源码安装。这里推荐使用 `pip` 安装二进制包³，`pip` 为 8.1 以上版本。过程非常简单，只需要短短几行命令即可。

```
$ easy_install pip
$ pip install -U six
$ pip install tensorflow
```

通过 `pip` 安装的库默认位置为 `/usr/local/lib/python2.7/site-packages`。

对于配备独立显卡的 Mac 电脑，可以安装 GPU 版本：

```
$ pip install tensorflow-gpu
```

需要注意的是，若要启用 GPU 支持，需要先安装 `coreutils` 和 `cuda` 库。安装同样使用 `brew`，详细步骤可查看官方文档。

顺利执行结束后，可以检测一下安装：

```
$ python
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> sess.run(hello)
Hello, TensorFlow!
>>>
```

搞定！就是这么简单！

3.1.2 Linux GPU 服务器安装

对于一般的深度学习程序来说，目前最具综合优势的运行设备非 GPU 莫属。Nvidia 作为当今世界著名的显卡生产制造企业（网友戏称为核弹厂，因为出品过多款性能强劲但功耗和发热量都极高的“核弹级”显卡），已经早早下注深度学习领域，全力开发高性能计算显卡。在最新的评测中，几款最强显卡在关键程序上的计算性能

3 推荐使用阿里 `pip` 源：<http://mirrors.aliyun.com/help/pypi>。

指标是 Intel E5 系列服务器 CPU 的 10 倍以上。也就是说, 10 台 CPU 服务器全力运行, 才能抵得上一块高性能计算显卡, 差距十分明显。也正因如此, 在深度学习的研究和应用方面, 高性能 GPU 服务器可以说是必不可少的基础设施。

要拥有自己的 GPU 服务器可以有两种方式, 一种是直接购买配件自行搭建服务器, 另一种是购买云计算服务商提供的 GPU 云主机。国内外比较知名的云计算厂商, 如 Amazon、阿里云、UCloud 等, 都提供了 GPU 云服务器的选择。而搭建自己的服务器能更灵活地满足业务需求, 并且从目前的价格来说, 也更加经济实惠。所以, 在有条件的情况下, 组装服务器是目前更推荐的选择。

首先来介绍最为关键的 GPU 的选择。对于科学计算而言, 决定性能最关键的指标是 GPU 的浮点运算能力。特别地, 对于目前主流的深度学习应用来说, 单精度浮点运算能力是核心需求。所以, 在单精度浮点运算方面综合性价比最高的就是合理的选择。从这方面讲, Tesla 系列虽然性能出众, 但由于价格过于昂贵, 对于深度学习应用来说优势不太明显。Geforce GTX 系列一直是 Nvidia 在游戏市场的主打, 拥有较高的性价比。其中, Titan X 是计算单元最多、性能最高的一款产品, 在深度学习研究中被广泛应用。在配置服务器时, 推荐选择同系列显卡中性能最高的产品⁴, 原因是如果要达到同等的计算能力, 顶级显卡需要更少的机器数量。否则, 不仅会导致 CPU、内存、主板、硬盘等配套硬件的额外成本, 还会增加维护的成本。

其次, 服务器的供电和散热是需要重点关注的方面。Nvidia 的高性能显卡素来以高功耗、高发热量著称, 若一台服务器装满 4 块或 8 块显卡, 那耗电量和发热量都是相当高的。所以, 为主机配备稳定的电源, 甚至 UPS 等保证供电稳定的设备, 配合强劲的散热风扇, 对保护服务器都有巨大的帮助。

最后, 除了 GPU 以外的其他部分, 如 CPU、内存、硬盘、网络带宽等, 没有特别要求, 达到普通服务器的平均水平就基本不会构成性能瓶颈。如果有大数据的 I/O 操作, 附加配置一块 SSD 固态硬盘即可满足。

服务器的操作系统推荐选择 Ubuntu 或 Cent OS, 稳定、配套全面、升级方便,

4 Nvidia 官方给出了各种 GPU 计算性能的参照表, 可以参看网址: <https://developer.nvidia.com/cuda-gpus>。

总之就是省力省心。

服务器初步搭建完毕后，需要安装 CUDA Toolkit 和 cuDNN 库以启用 GPU 加速环境。CUDA Toolkit 是 Nvidia 提供的 CUDA 开发套件，其中包括了 CUDA 驱动，以及编写 CUDA 程序必须使用的编译器和头文件，还有一些辅助函数库。cuDNN 是基于 CUDA Toolkit 编写的专门面向深度神经网络的 GPU 加速库，其中提供了一系列深度神经网络常用的计算，如网络的 forward 和 backward 计算、卷积计算、池化计算、标准化，以及多种激活函数，TensorFlow 直接使用这些库函数来实现各种计算的 GPU 加速。

CUDA Toolkit 直接从 Nvidia 官方网站上选择相应平台下载二进制安装文件即可⁵。以 CUDA 8.0 为例，下载文件为 `cuda_8.0.44_linux.run`，使用 Bash 命令执行安装，默认安装位置为 `/usr/local/cuda`：

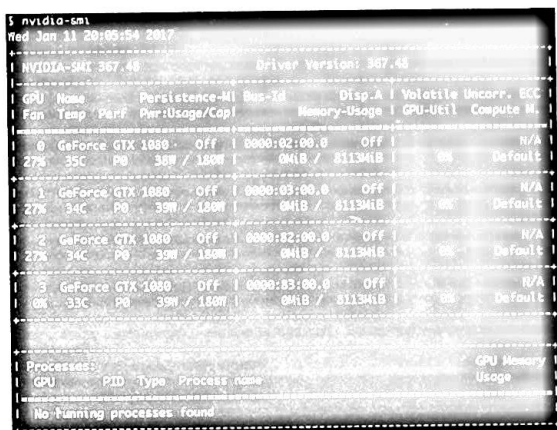
```
$ sudo bash cuda_8.0.44_linux.run
```

安装完成后可以发现，Nvidia 提供了多种辅助工具。其中，`nvidia-smi` 命令可以查看服务器上每块显卡的状态。可以从图 3-1 中看到，`nvidia-smi` 命令显示了服务器上每一块显卡的信息，包括型号、温度、功率、显存使用情况、CUDA 计算单元的使用情况等，还能看到哪些进程正在使用 GPU 执行计算。这些信息对程序剖析有一定的帮助。

TensorFlow 对于神经网络的 GPU 加速都通过 cuDNN 库实现。cuDNN 可以在 Nvidia 官方网站下载⁶，需要注意的是，下载版本需要与 CUDA Toolkit 的版本相匹配。cuDNN 库以动态链接库的形式提供，解压下载 tar 包，并设置 `LD_LIBRARY_PATH` 环境变量使系统可以加载，或者将环境变量设置在 `bashrc` 文件中会更加方便。

5 CUDA Toolkit 下载网址：<https://developer.nvidia.com/cuda-downloads>。

6 cuDNN 下载网址：<https://developer.nvidia.com/cudnn>。



```

$ nvidia-smi
Wed Jan 11 20:05:54 2017

+-----+
| NVIDIA-SMI 367.48                  Driver Version: 367.48 |
+-----+-----+
| GPU Name           Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|  Memory-Usage | GPU-Util  Compute M. |
+-----+-----+-----+-----+
| 0  GeForce GTX 1080  Off      | 0000:02:00:0 | Off |         N/A |
| 27%   35C   P0     36W / 160W |  6MiB / 1134MiB |   0%   Default |
+-----+-----+-----+-----+
| 1  GeForce GTX 1080  Off      | 0000:03:00:0 | Off |         N/A |
| 27%   34C   P0     39W / 160W |  6MiB / 1134MiB |   0%   Default |
+-----+-----+-----+-----+
| 2  GeForce GTX 1080  Off      | 0000:02:00:0 | Off |         N/A |
| 27%   34C   P0     39W / 160W |  6MiB / 1134MiB |   0%   Default |
+-----+-----+-----+-----+
| 3  GeForce GTX 1080  Off      | 0000:03:00:0 | Off |         N/A |
| 0%    33C   P0     39W / 160W |  6MiB / 1134MiB |   0%   Default |
+-----+-----+-----+-----+

Processes:
+-----+-----+-----+-----+
| GPU    PID  Type  Process name                        | GPU Memory |
|-----+-----+-----+-----+
| No running processes found. |
+-----+-----+-----+-----+

```

图 3-1 用 nvidia-smi 工具查看显卡信息

```

$ cd <install_path_of_cudnn>
$ export LD_LIBRARY_PATH=`pwd`:LD_LIBRARY_PATH

```

安装完 CUDA Toolkit 和 cuDNN 库后, 同样使用 pip 来安装 TensorFlow。

```

$ sudo apt-get install python-pip python-dev
$ sudo pip install --upgrade pip
$ sudo pip install tensorflow-gpu

```

检测一下安装:

```

$ python
>>> import tensorflow as tf
I tensorflow/stream_executor/dso_loader.cc:108] successfully
opened CUDA library libcublas.so locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully
opened CUDA library libcudnn.so locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully
opened CUDA library libcufft.so locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully
opened CUDA library libcuda.so locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully
opened CUDA library libcurand.so locally
>>> tf.__version__
'0.12.1'
>>>

```

可以看到, 与 CPU 版本不同的是, GPU 版本的 TensorFlow 在启动时会加载 CUDA

相关的动态链接库，如 `libcuda.so`、`libcudnn.so` 等。在执行时看到这样的 log 就说明 GPU 版本 TensorFlow 安装成功了。

3.1.3 常用 Python 库

MATLAB 曾经是机器学习领域里的主流语言，是因为它拥有强大的科学计算能力和丰富的数据处理库。而在当下，Python 得到了越来越多的数据分析专业人士的青睐，是因为它不但拥有灵活强大的语法，有较低的学习门槛，而且出现了 NumPy 等专门用于科学计算的函数库，补足了计算性能上的问题。配合以更多辅助工具库，如绘制图形的 Matplotlib 等，让 Python 成为当前在机器学习领域使用最多的语言。

这里介绍几个常用工具库。

1. NumPy

NumPy 库为 Python 提供了基础的科学计算能力，遵循 BSD 开源协议。在标准的 Python 中，为了保持数组对象的动态特性，数组中实际存储的是每个元素的指针，存储和访问时都需要经过多次指针跳转。这样的方式虽然非常灵活，但对于数值运算来说既浪费内存又低效。NumPy 弥补了这方面的不足，它提供了如 C 语言一样的高效的 N 维数组结构 `ndarray` (N-dimensional array object) 和一系列直接对数组进行处理的函数 `ufunc` (universal function object)，不但保留了 Python 简洁灵活的语法，也带来了不逊于 C 语言程序的运算速度。除此之外，NumPy 还提供了常用的线性代数计算，甚至傅里叶变换等函数，可以进行多种信号处理。

2. pandas

pandas 全称为 Python Data Analysis Library，是一个高性能数据结构和数据分析工具，同样遵循 BSD 开源协议。pandas 基于 NumPy 构建，让以 NumPy 为中心的应用开发变得简单。它提供了一种高效的 `DataFrame` 结构，可以自动对齐、补全数据，免去了由于输入数据缺失导致的问题。还可以灵活地完成增加、删除数据列，调整列的顺序等元数据操作。此外，`DataFrame` 还能像数据库一样进行一些简单的查询、聚合操作。因此，可以说 pandas 是解决数据处理问题不可或缺的重要工具之一。

3. Matplotlib

Matplotlib 是功能强大的画图引擎，可以制作高质量的图表。绘图是数据分析工作中的一项重要任务，通过简洁易懂的类 MATLAB 接口，可以只用短短几行代码绘制曲线图、散点图、直方图、柱状图、饼状图等图标，方便人们更加直观地感受数据，并且完成分析数据的特征、查找异常数据值等工作。另外，通过 pandas 集成的部分工具函数，如 `scatter_matrix`，可以直接对高维数据进行可视化观察，对于分析数据处理数据有非常重要的帮助。

4. PIL

PIL 全称 Python Imaging Library，目前已经成为 Python 平台事实上的标准图像处理库。PIL 可以方便地读入和输出包括 jpg、png 等多种常见类型的图像文件，还能对图像做切割、翻转、添加文字等变换。是处理图像数据的常用工具之一。

5. IPython & Jupyter

Jupyter 是一个开源的交互式数据分析处理平台。Jupyter Notebook 能以 Web 网页的形式创建和分享文档，并可以在文档中插入代码段，交互式地查看代码运行结果。IPython 是 Jupyter 的前身，也是目前 Jupyter 中 Python 代码的执行引擎。目前，Jupyter 已经支持包括 R、Julia 在内的多种数据分析常用语言。同时，因其搭建服务的便捷性，加上多种扩展功能，如制作精美的幻灯片，使其成为分享代码、分享文档的利器。

6. scikit-learn (sklearn)

scikit-learn 的标语是“Machine Learning in Python”，是构建在 NumPy、Matplotlib 等工具之上的一套完整的机器学习工具库。在该库中封装了多种常用的分类、回归、聚类、数据降维、数据预处理等算法，三五行代码就可以完成简单机器学习模型的定义和训练，使数据挖掘变得非常简单。scikit-learn 的接口设计是如此的合理易用，以至于 TensorFlow 出现以后，很多项目都仿照 scikit-learn 的接口对 TensorFlow 进行了二次封装，使接口更加简洁，代码更加明晰易读。

7. OpenCV

OpenCV 的全称是 Open Source Computer Vision Library，是一款跨平台机器视觉

工具库,在产品质量检测、医学成像、机器人、监控摄像机定位等多种计算机视觉应用领用中都有使用。OpenCV 内核使用 C/C++ 语言开发,运行速度快。前端支持 Python、MATLAB、Ruby 等多种语言。OpenCV 中包含图像滤波、特征提取、视频分析、三维重建等功能模块,还包括人脸识别、目标检测等高级功能,可以满足许多图像处理方面的高级需求。此外,由于计算机视觉与机器学习密切相关,OpenCV 中还集成了机器学习库,可以用于处理一些模式识别和聚类方面的问题。

在这些功能强大的工具的辅助下,数据分析处理的工作就可以事半功倍。在本书的实际应用例子中,会穿插介绍这些工具中的部分功能。

3.2 Titanic 题目实战

实践是最好的学习手段。接下来我们通过尝试用 TensorFlow 解决一个分类问题,了解 TensorFlow 的编程模式和常用接口,以及一些提升预测准确率的优化思路。

Titanic 问题是 Kaggle 平台上的一个练手题目⁷,要求使用数据分析的方法预测泰坦尼克号的幸存者名单。虽然这并不是一个有奖金的正式题目,仅供大家尝试和学习,但是由于数据简单整洁,又是典型的二分类问题,作为第一次实践是非常不错的题目。

3.2.1 Kaggle 平台介绍

Kaggle 平台是著名的数据分析竞赛在线平台,创始于 2010 年。在 Kaggle 平台上有各种有趣的数据挖掘挑战题目,包括分类、预测、推荐等多种类型。全世界的科学家和数据分析师可以以组队或者 solo 的形式参与竞赛。参赛者可以使用任何方法对题目给出的数据进行分析,并对测试集数据进行预测,最后平台以预测准确率为标准判定参赛者的成绩。一些知名企业和机构也会与 Kaggle 平台合作,将业务中的一些难题提出为数据挖掘任务,并设立高额奖金,让更多的专家参与解决,比如针对高清卫星图的物体识别、产品推荐、超声图像识别、鱼类检测识别,等等。

Kaggle 作为竞赛平台,拥有一套比较完备的评分系统。参赛者的成绩一般以预测的准确率排名,预测结果越准确则排名越高。参赛者每天的排名设立了公开排行榜

⁷ Titanic 题目网址: <https://www.kaggle.com/c/titanic>。

(public leaderboard) 和非公开排行榜 (private leaderboard)。在公开排行榜中, 所有人随时都可以查看预测准确率和当前排名, 不过这里对于准确率的计算并不是以全部测试数据作为基准, 而只是测试集中的一部分数据, 或者说可以认为是平台的验证集数据。公开榜可以让参赛者对自己算法的表现有一个初步评估, 通过与他人对比来看是否还有提升空间。为了防止参赛者利用多次提交的方法来试探测试集数据的结果, 人肉对测试数据“过拟合”, 竞赛最终成绩由另一份测试数据, 也就是非公开排行榜的排名所决定的。

除了举办各种比赛之外, Kaggle 平台上还积累了各种有趣的数据集⁸。其中被关注较多的有欧洲足球赛对战数据, 在这份数据中囊括了超过 25000 场比赛和超过 10000 名球员的数据, 其中被详尽记录 (进球、角球、越位、红黄牌等) 的比赛就超过 1 万场, 甚至有 10 家博彩机构给出的投注赔率。通过这样的大数据不仅可以对各个球员或球队的历史进行多维度分析, 计算各种单项和综合实力, 更能对未来的表现进行预测。这样丰富全面数据确实不可多得。

Kaggle 平台是一个可以让人们充分交流的地方, 参赛者们不仅比拼技术, 更能交流经验。即使只是翻看各个题目的讨论帖, 也会觉得受益匪浅, 例如题目会有数据图形化讨论专区, 在里面会从各种维度给出数据的可视化视图。这对于初学者来说是非常好的学习途径。

3.2.2 Titanic 题目介绍

泰坦尼克号轮船 (RMS Titanic, 又称为铁达尼号) 的沉没是历史上最著名的海难事故之一。作为当时世界上最大的客运轮船, 它在处女航中由于撞上冰山导致轮船断裂, 并最终沉入大西洋。当时船上共有 2224 名乘客和船员, 其中 1502 名丧生。在回顾这样一场浩劫的过程中, 分析发现造成如此巨大损失的原因之一是当时轮船上并没有准备足够的救生艇。在这起事件发生后, 除了推动建立了更合理的航行安全规范外, 人们还分析了当时对于仅有的救生艇位置的分配过程, 虽然最后幸存是有一定的运气成分, 但一些乘客确实比另外一些更容易获救, 比如妇女、儿童, 以及高级舱位的乘客。

⁸ Kaggle 数据库: <https://www.kaggle.com/datasets>。

本题目是希望重新站在客观的视角来重新回顾这一事件。通过分析事发时的数据，使用机器学习的方法来预测哪些乘客最终能在这次灾难中幸存下来。更具体来说，是通过乘客的各项信息，如姓名、性别、年龄、乘船客舱等级等信息，尝试预测每位乘客幸存的概率。

既然是要预测每位乘客是否可以幸存，那么可以认为这是一个二分类问题，即一类标签为存活，另一类标签为丧生。我们最终需要训练一个分类器，可以是 SVM、神经网络、随机森林等模型，来判定样本所属的类别。

对于此类数据挖掘题目，对数据进行一些简单的分析和处理是必要的第一步。从 Kaggle 网站下载本问题的数据集⁹，可以看到，数据分为训练集和测试集两类，均以 csv 格式存储。训练集数据文件 train.csv 大小为 59.76KB，包含 891 条数据，测试集数据文件 test.csv 的大小为 27.96KB，包含 418 条数据。在数据文件中，每一行是一个样本，代表一名乘客的信息，包含以下 12 个字段，如表 3-1 所示。

表 3-1 Titanic 挑战字段说明

字段名称	字段解释	类 型	说 明
PassengerId	乘客 ID	int	训练集数据 ID 编号为 1~891，测试集数据编号为 892~1309
Survived	是否幸存	bool	0：故去，1：幸存
Pclass	客舱等级	int	客舱分为 3 级，1 为头等舱，2 为中等舱，3 为普通船舱
Name	乘客姓名	string	
Sex	乘客性别	string	"male"：男性，"female"：女性
Age	乘客年龄	int	年龄为整数
SibSp	兄弟姐妹和配偶在船数量	int	
ParCh	父母孩子的在船数量	int	
Ticket	船票编号，例 "STON/O2. 3101282"	string	
Fare	船票价格	float	
Cabin	客舱位置，例 "C123"	string	
Embarked	登船港口的编号，例 "S"	string	C 代表 Cherbourg； Q 代表 Queenstown； S 代表 Southampton

⁹ Titanic 问题数据下载网址：<https://www.kaggle.com/c/titanic/data>。

其中,“Survived”字段代表该名乘客最终是否生还,是样本的标签字段。图 3-2 展示了前 10 条训练数据的内容:

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cummings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.925		S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1	0	113803	53.1	C123	S
5	0	3	Allen, Mr. William Henry	male	35	0	0	373450	8.05		S
6	0	3	Moran, Mr. James	male		0	0	330877	8.4583		Q
7	0	1	McCarthy, Mr. Timothy J	male	54	0	0	17463	51.8625	E46	S
8	0	3	Palsson, Master. Gosta Leonard	male	2	3	1	349909	21.075		S
9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27	0	2	347742	11.1333		S
10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14	1	0	237736	30.0708		C

图 3-2 Titanic 挑战前 10 条样本数据

测试集的每条数据包含了除“Survived”字段以外的其他所有字段。

最后需要输出的结果包含两列,分别为测试集乘客 ID 和对应乘客是否幸存预测。

通过观察题目给定的条件,我们可以根据一些生活常识和对 Titanic 事件的了解,对数据进行一些初步推断。比如我们看过《泰坦尼克号》电影的都知道,事故发生在寒冷的海上冰山,如果在轮船沉没前登上了为数不多的救生筏,则基本可以幸存,否则落入寒冷的大西洋就一定凶多吉少,就像电影中的男主人公 Jack 一样。那么在这样一种情况下,都有哪些人登上了救生筏呢? 首先是妇女和儿童,轮船上大部分人为英国人,英国是讲究绅士风度的国家,不管是船员还是乘客都会一致认为应该让妇女和儿童先登上救生筏,也因此妇女儿童幸存的比例会比成年男性要高。其次,高等客舱的乘客基本是贵族、军官等有身份地位的人,他们在事故发生后也会得到额外照顾,获得优先登上救生筏的资格。剩下的其他人就没有那么幸运了,幸存概率比较低。这些先验知识,在后续的判定中将起到非常大的作用。

对于我们认为相关性较高的字段,比如客舱等级“Pclass”、乘客性别“Sex”、乘客年龄“Age”,将在之后作为主要的特征字段,需要经过正规化(Normalization)处理后转换为数值形式。在本例中,正规化主要有三个方面,一是将字符串字段转换为数值化的表达方式,比如将性别字段原本取值“male”和“female”分别转换为 0 和 1,这样才能作为分类器的输入;二是可以将数值都归一化到 [0,1] 的取值范围内,比如年龄字段原本的值域是 [0,100],归一化过程可以是将每个值都除以 100;三是补齐缺失数据。

除了我们认为的关键字段以外，其他字段可能也包含很多有用的信息，部分可以经过挖掘转换后作为特征使用。对于一些现实生活中的复杂问题，人类的认知可能也是片面的、不准确的，真实的相关性可能隐藏在其他数据中，所以抓取更多数据是惯用的思路，但同时会随之出现维度灾难（curse of dimensionality）的问题，也就是维度过多、无效的噪声过多，导致无法有效地从中过滤出真正有用的信息。在本例中，如姓名“Name”、父母子女在船舱数量“ParCh”、客舱位置“Cabin”、登船码头编号“Embarked”等几个字段，猛一看似乎与能否逃生并没有太大的直接关系，但若经过挖掘也许会有意外的收获。在本章后半部分会进一步深入讨论。

经过上述分析后，我们可以得出解决 Titanic 问题的主要思路，即首先采用正规化操作等手段对原始数据进行预处理，然后挑选特征向量的维度，并以此训练一个分类器，最终使用训练好的分类器来预测测试集数据的结果。

接下来，就可以进入真正的编码解题阶段。作为首次尝试，无须追求一步到位地做到最好，而应该先以简单规则得出基线（baseline）版本，然后不断优化以得到更好的结果。因此，我们首先尝试用 TensorFlow 训练一个逻辑回归分类器来看看效果如何。

整个代码可以分为：数据读入及预处理，构建计算图，构建训练迭代过程，执行训练、存储模型，预测测试数据结果几个部分。下面就对每一部分分别加以介绍¹⁰。

3.2.3 数据读入及预处理

对数据进行预处理会使用到 pandas 和 scikit-learn 库所提供的一些功能。

首先，使用 pandas 内置的数据文件解析器读入数据。pandas 内置了多种解析器，可以直接处理 csv、pickle、json、excel、html 等常用的数据文件格式，甚至还可以从 MySQL 数据库或者操作系统的剪切板中读入数据。读入操作非常简单，只需要用 read_csv() 函数读取“train.csv”文件，读入的数据为一个 DataFrame 类型的对象。

```
import pandas as pd
```

```
# 从 CSV 文件中读入数据
```

¹⁰ 完整版本代码可参见：<https://github.com/wangchen1ren/Titanic>。

```
data = pd.read_csv('train.csv')
```

DataFrame 是一个类似于电子表格的二维数据结构，长度可变，维度可变，类型亦可变。在 DataFrame 中，行列都经过排序编号。

可以通过 DataFrame.info() 方法查看数据的概况：

```
>>> data.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId      891 non-null int64
Survived         891 non-null int64
Pclass           891 non-null int64
Name             891 non-null object
Sex              891 non-null object
Age              714 non-null float64
SibSp            891 non-null int64
Parch            891 non-null int64
Ticket           891 non-null object
Fare             891 non-null float64
Cabin            204 non-null object
Embarked         889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.6+ KB
```

可以看到，数据包含 891 条记录，下标编号为 0 到 890。数据包含 12 个列，正如前面介绍过的一样。其中，PassangerId、Survived、Pclass、Age、SibSp、Parch、Fare 几个字段为数值类型（64 位整形或 64 位浮点型），而 Name、Sex、Ticket、Cabin、Embarked 字段是对象，其实也就是字符串类型。同时，还可以从这个简短的统计看到，Age、Cabin、Embarked 三个字段存在缺失的情况，Age 字段只有 714 个有效值，缺失 177 个；Embarked 有 889 个值，缺失 2 个；Cabin 字段仅有 204 个值，缺失达到 687 个。

为了方便处理，我们仅保留 Sex、Age、Pclass、SibSp、Parch、Fare 这 6 个字段。在前文中提到，首先对各字段做正规化处理。具体来说，要将 Sex 字段的字符串进行转换，将“male”替换为 0，将“female”替换为 1。同时，将 Age 字段有缺失的部分统一赋值为 0。

另外，需要将样本的标签转换为独热编码 (one-hot encoding)。独热编码又称为一位有效编码，是分类标签常用的编码方式。具体来说，就是使用 N 位布尔型的状态标识来对 N 种状态进行编码，任意时刻编码中只有一位有效，即取值为 `True`。在本例中，`Survived` 是幸存一类的标签，新建一个 `Deceased` 字段来表示乘客是否死亡，其取值为 `Survived` 的取非。这样 `Survived` 与 `Deceased` 两个字段一起相当于构成了一组 one-hot 编码。

```
# 取部分特征字段用于分类，并将所有缺失的字段填充为 0
data['Sex'] = data['Sex'].apply(lambda s: 1 if s == 'male' else 0)
data = data.fillna(0)
dataset_X = data[['Sex', 'Age', 'Pclass', 'SibSp', 'Parch', 'Fare']]
dataset_X = dataset_X.as_matrix()

# 两种分类分别是幸存和死亡，'Survived' 字段是其中一种分类的标签，
# 新增 'Deceased' 字段表示第二种分类的标签，取值为 'Survived' 字段取非
data['Deceased'] = data['Survived'].apply(lambda s: int(not s))
dataset_Y = data[['Deceased', 'Survived']]
dataset_Y = dataset_Y.as_matrix()
```

然后，为了防止训练过拟合，我们将仅有的标记数据分成训练数据集 (training dataset) 和验证数据集 (validation dataset) 两类。验证数据不参与模型训练，样本数占全部标记数据的 20%。scikit-learn 库中提供了用于切分数据集的工具函数 `train_test_split()`，随机打乱数据后按比例拆分数数据集。

```
from sklearn.model_selection import train_test_split

# 使用 sklearn 的 train_test_split 函数将标记数据切分为“训练数据集和验证数据集”
# 将全部标记数据随机洗牌后切分，其中验证数据占 20%，由 test_size 参数指定
X_train, X_test, y_train, y_test = train_test_split(
    dataset_X, dataset_Y, test_size=0.2, random_state=42)
```

3.2.4 构建计算图

逻辑回归是形式最简单，并且最容易理解的分类型之一。从数学上，逻辑回归的预测函数可以表示为如下公式：

$$y' = \text{softmax}(xW + b)$$

其中, x 为输入向量, 是大小为 $d \times 1$ 的列向量, d 是特征数。 W 是大小为 $c \times d$ 的权重矩阵, c 是分类类别数目。 b 是偏置向量, 为 $c \times 1$ 列向量。在数学定义里, softmax 是指一种归一化指数函数。它将一个 k 维的向量 z 按照下列公式

$$\delta(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

的形式将向量中的元素转换为 $(0, 1)$ 的区间。机器学习领域常使用这种方法将类似判别函数的置信度值转换为概率形式 (如判别超平面的距离等)。softmax 函数常用于输出层, 用于指定唯一的分类输出。

权重矩阵 W 和偏置向量 b 是模型中的参数, 也就是要通过训练来求得的部分。

使用 TensorFlow 构建这样的计算过程是十分简单的, 整个构建计算图的过程只需要以下几个步骤。

1. 使用 placeholder 声明输入占位符

在第 2 章中曾经介绍过, TensorFlow 设计了数据 Feed 机制。也就是说计算程序并不会直接交互执行, 而是在声明过程只做计算图的构建。所以, 此时并不会触碰真实的数据, 而只是通过 placeholder 算子声明一个输入数据的占位符, 在后面真正运行计算时, 才用数据替换占位符。

声明占位符 placeholder 需要给定三个参数, 分别是输入数据的元素类型 dtype、维度形状 shape 和占位符名称标识 name。TensorFlow 内置了所有标准数据类型, 包括 int16、uint16、int32、uint32、float16、float32、string 等, 在指定元素类型时可以直接使用。维度形状使用数组指定。若 shape 不指定, 默认值为 None, 表示任意形状。需要注意的是, 通过 mini-batch 批量训练是如今常用的优化技巧, 通常能在更短时间内得到更好的拟合效果。所以在定义输入形状时, 一般将第一个维度作为 mini-batch 维度, 而从第二个维度开始才是特征维度。占位符名称 name 用于区分计算图里的各个节点, 不管在查找节点, 还是在可视化方面, 设置容易识别的名称都非常有帮助。名称参数 name 默认也为 None, 系统会自动将节点设置为类似 “Placeholder:0” 这样的名称。

本例中, 对于模型来说有两个输入数据, 一个为特征数据 X , 由 Sex、Age、Pclass、

SibSp、Parch、Fare 这 6 个字段的值组成，另一个为标记值 y ，由 Deceased、Survived 两个字段组成。

```
# 声明输入数据占位符
# shape 参数的第一个元素为 None，表示可以同时放入任意条记录
X = tf.placeholder(tf.float32, shape=[None, 6])
y = tf.placeholder(tf.float32, shape=[None, 2])
```

2. 声明参数变量

逻辑回归模型中包含两个参数，分别是权重矩阵 W 和偏置向量 b 。TensorFlow 使用变量 (Variable) 来存储和更新这些参数的值。

变量的声明方式是直接定义 `tf.Variable()` 对象。初始化变量对象有两种方式，一种是从 protocol buffer 结构 `VariableDef` 中反序列化，另一种是通过参数指定初始值。最简单的方式就是像下面程序这样，为变量传入初始值。初始值必须是一个 tensor 对象，或是可以通过 `convert_to_tensor()` 方法转换成 tensor 的 Python 对象。TensorFlow 提供了多种构造随机 tensor 的方法，可以构造全零 tensor、随机正态分布 tensor 等。定义变量会保留初始值的维度形状。

值得注意的是，变量 `Variable` 的构造函数同样也没有做数据的初始化，而是在计算图中加入了一个 `variable` 算子和其对应的 `assign` 算子。

```
# 声明变量
W = tf.Variable(tf.random_normal([6, 2]), name='weights')
b = tf.Variable(tf.zeros([2]), name='bias')
```

3. 构造前向传播计算图

所谓前向传播就是网络正向计算，由输入计算出标签的过程。逻辑回归的公式用 TensorFlow 表示只需要一行代码：

```
y_pred = tf.nn.softmax(tf.matmul(input, W) + bias)
```

其中，`tf.matmul()` 是矩阵乘法算子，`tf.nn.softmax()` 是 softmax 函数。另外可以看到，对于偏置向量 `bias`，可以直接用加号“+”完成矩阵相加操作，这与 NumPy 等库用法类似。TensorFlow 中的 Tensor 对象和 Variable 对象都对常用四则运算符号进行过重载，运用十分灵活。

在计算图的构建过程中，TensorFlow 会自动推算每一个节点的输入输出形状。若无法运算，比如两个行列数不同的矩阵相加，则会直接报错。

4. 声明代价函数

机器学习算法的优化需要靠代价函数来评估优化方向。本例的二分类问题一般使用交叉熵（cross entropy）作为代价函数。

交叉熵的计算公式为：

$$C = -\frac{1}{n} \sum_x y \log(y')$$

程序代码为：

```
# 使用交叉熵作为代价函数
cross_entropy = - tf.reduce_sum(y * tf.log(y_pred + 1e-10),
                                reduction_indices=1)
# 批量样本的代价值为所有样本交叉熵的平均值
cost = tf.reduce_mean(cross_entropy)
```

值得注意的是，在计算交叉熵的时候，对模型输出值 `y_pred` 加上了一个很小的误差值（在上面程序中是 `1e-10`），这是因为当 `y_pred` 十分接近真值 `y_true` 的时候，也就是 `y_pred` 的值非常接近 0 或 1 时，计算 `log(0)` 会得到负无穷 `-inf`，从而导致输出非法，全部都是 `nan`，并进一步导致无法计算梯度，迭代陷入崩溃。要解决这个问题有三种办法：

- 在计算 `log()` 时，直接加入一个极小的误差值，使计算合法。这样可以避免计算 `log(0)`，但存在的问题是加入误差后相当于 `y_pred` 的值会突破 1。在示例代码中使用了这种方案；
- 使用 `clip()` 函数，当 `y_pred` 接近 0 时，将其赋值成为极小误差值。也就是将 `y_pred` 的取值范围限定在 $[10^{-10}, 1]$ 的范围内；
- 当计算交叉熵的计算出现 `nan` 值时，显式地将 `cost` 设置为 0。这种方式回避了 `log()` 函数计算的问题，而是在最终的代价函数上进行容错处理。

5. 加入优化算法

TensorFlow 内置了多种经典的优化算法，如随机梯度下降算法（Stochastic Gradient Descent, SGD）、动量算法（Momentum）、Adagrad 算法、ADAM 算法、RMSProp 算法，另外还有在线学习算法 FTRL。优化器内部会自动构建梯度计算和反向传播部分的计算图。

一般对于优化算法，最关键的参数是学习率（learning rate），对于学习率的设置是一门技术。同时，不同优化算法在不同问题上可能会有不同的收敛速度，在解决实际问题时可以做多种尝试。

```
# 使用随机梯度下降算法优化器来最小化代价，系统自动构建反向传播部分的计算图
train_op
tf.train.GradientDescentOptimizer(0.001).minimize(cost)
```

至此，计算图的声明过程就完成了。

3.2.5 构建训练迭代过程

完成了计算图的定义，再接下来要构建训练迭代。

在第 2 章中提到过，在 TensorFlow 的设计中，前端编程语言要触发后端执行引擎开始计算，必须通过 Session 接口完成。Session 对象负责将运行环境打包，并且管理运行时需要处理的变量、队列（queues）、读取器（readers）等资源。

需要特别说明的是，由于 Session 中管理了上下文的各种资源，所以在计算执行结束后一定要关闭，以释放对资源的占用。一般有两种使用方式，一种是在声明后手动调用 Session.close() 方法来关闭，一般会在大型应用中使用。另外一种是为更简便的方式，Session 类重载了 __enter__() 和 __exit__() 两个方法，所以可以用 Python 的 with 语句将 Session 作为上下文管理器（Context Manager）来操作，退出作用域时自动关闭对象。

Session 启动后就正式进入了训练过程。首先要做的是使用 tf.global_variables_

`initializer().run()` 方法初始化所有的变量¹¹。接下来就是用一个循环将训练数据反复代入计算图来执行迭代。在循环内，`Session.run()`是触发后端执行的入口。

`Session.run()` 有两个关键的参数，`fetches` 和 `feed_dict`。其中，`fetches` 指定需要被计算的节点，可以用数组同时制定多个节点。节点可以是算子 `op`，比如前文程序中的优化算法算子 `train_op`，也可以是 `tensor`，比如代表代价函数值的 `cost`。执行会从输入节点开始，按照节点的依赖关系，也就是有向图的拓扑序，依次计算目标节点所在的子图中的所有节点。计算所需要的输入数据则由 `feed_dict` 代入。`feed_dict` 需要传入一个字典，字典的 `key` 是输入占位符 `placeholder`，`value` 为真实的输入数据。

`Session.run()` 执行完计算后，会返回计算节点的结果。若节点为算子，则没有返回值，若节点是 `tensor`，则返回当前的值。比如最常见的优化算子和代价函数 `tensor` 的组合，而优化算子的执行结果其实是通过梯度下降算法更新所有参数变量，不需要返回值，而在前向传播计算得到的代价值将会返回。

通过将每一轮迭代的代价值打印出来，可以监控训练的收敛情况。

```
with tf.Session() as sess:
    # 初始化所有变量，必须最先执行
    tf.global_variables_initializer().run()

    # 以下为训练迭代，迭代 10 轮
    for epoch in range(10):
        total_loss = 0.
        for i in range(len(X_train)):
            feed = {X: [X_train[i]], y_true: [y_train[i]]}
            # 通过 session.run 接口触发执行
            _, loss = sess.run([train_op, cost], feed_dict=feed)
            total_loss += loss
        print('Epoch: %04d, total loss=%.9f' % (epoch + 1,
            total_loss))
    print 'Training complete!'
```

训练迭代循环执行结束后，用验证数据集评估模型的表现。

评估校验数据集上的准确率

¹¹ 注：TensorFlow 0.12.0 之前的版本使用 `tf.initialize_all_variables().run()`，在最新版本代码中标明该接口将于 2017 年 3 月 2 日后正式废弃。

```

pred = sess.run(y_pred, feed_dict={X: X_val})
correct = np.equal(np.argmax(pred, 1), np.argmax(y_val, 1))
accuracy = np.mean(correct.astype(np.float32))
print("Accuracy on validation set: %.9f" % accuracy)

```

从上面代码中可以看到，若只计算模型预测值 `y_pred`，则只需要传入占位符 `X` 所对应的数据即可，不需要指定另一个占位符 `y`。TensorFlow 在计算时会进行图的剪枝优化，只会计算 `fetches` 指定的子图，因此也只需要代入子图包含的占位符。在这一点上，TensorFlow 保证了编程的灵活性和执行效率，在使用时不用担心会执行不必要的计算。

3.2.6 执行训练

将代码保存为 "01_tensorflow_basic.py" 文件，通过命令行运行程序即可看到训练过程中的输出。

```

$ python 01_tensorflow_basic.py
Epoch: 0001, total loss=5827.214925724
Epoch: 0002, total loss=1866.215580815
Epoch: 0003, total loss=1416.462430373
Epoch: 0004, total loss=1274.361242647
Epoch: 0005, total loss=1287.683731217
Epoch: 0006, total loss=1207.134947834
Epoch: 0007, total loss=1191.541045362
Epoch: 0008, total loss=1180.067145067
Epoch: 0009, total loss=1169.640558892
Epoch: 0010, total loss=1160.065555358
Training complete!
Accuracy on validation set: 0.586592197

```

可以看到，通过 10 轮迭代的训练，代价值从 5827 下降到 1160，这代表模型对训练数据的拟合越来越好，误差越来越小。同时还可以看到，在验证数据集上，本次训练的预测正确率为 58%。

在执行程序的时候，可能会遇到的最常见的错误有如下几种：

- 变量未初始化

开启 `session` 后的第一步就需要初始化所有变量，否则在执行其他操作时将会产

生如下错误:

```
FailedPreconditionError: Attempting to use uninitialized value
Variable
```

解决方法是在 Session 对象初始化后,紧接着执行 `tf.global_variables_initializer().run()` 方法。

- 真实数据与占位符形状不匹配

如果在执行时出现类似下面这样的错误:

```
ValueError: Cannot feed value of shape (2,) for Tensor
u'Placeholder_1:0', which has shape '(?, 2)'
```

那一般是从 `feed_dict` 代入的真实数据的数组形状与占位符不匹配导致的。在确认传入数据无误的情况下,可以使用 `numpy.reshape()` 将数据调整成需要的形状后再代入计算。

- Session 已关闭

若在 Session 作用域之外执行运算,就会报出如下错误:

```
RuntimeError: Attempted to use a closed Session.
```

这时候就要检查确认 `Session.run()` 语句是在 Session 对象生命周期作用域内执行的。

3.2.7 存储和加载模型参数

训练是一件十分费时耗力的事情,如果每次预测之前都还要训练,那显然是不现实的。正确的姿势是在训练得到一组优秀的参数时将其保存下来,预测时直接加载到模型中使用。TensorFlow 当然也满足了这一需求,使用的是 `tf.train.Saver` 和 `checkpoint` 机制。

变量的存储和读取是通过 `tf.train.Saver` 类来完成的。Saver 对象在初始化时,为计算图加入了用于存储和加载变量的算子,并可以通过参数指定是要存储哪些变量。Saver 对象的 `save()` 和 `restore()` 方法是触发图中算子的入口。

Checkpoints 是用于存储变量的二进制文件，在其内部使用字典结构存储变量，键为变量名字，即 `Variable.name` 成员的值，值为变量的 `tensor` 值。TensorFlow 代码中提供了一个工具程序可以用于查看 checkpoint 文件的内容，代码在 `tensorflow/python/tools/inspect_checkpoint.py`。

Saver 最简单的用法就如下面代码所示。

```
v1 = tf.Variable(tf.zeros([200]))
saver = tf.train.Saver()
# 在 Saver 之后声明的变量将不会被 Saver 处理
v2 = tf.Variable(tf.ones([100]))

# 训练 Session 创建参数存档
with tf.Session() as sess1:
    # 完成模型训练过程
    ...
    # 持久化存储变量
save_path = saver.save(sess1, "model.ckpt")

# 在新 Session 中加载存档
with tf.Session() as sess2:
    # 加载变量
    saver.restore(sess2, "model.ckpt")
    # 判别预测，或继续训练
    ...
```

需要注意的是，Saver 对象在初始化时，若不指定变量列表，默认只会自动收集其声明之前的所有变量，在 Saver 对象初始化后的所有变量将不被记录，上面代码中的 `v2` 变量就不会在存取的范围之内。这样的机制在迁移学习的应用中非常有用，例如要将一个基于 ImageNet 数据训练好的 CNN 应用在新类型图片识别上，只需要加载模型卷积部分的参数，重新训练最后的全连接网络即可。

在上面的程序示例中，由 `Saver.save()` 触发的存储操作会生成 4 个文件¹²。第一个是名为“`model.ckpt`”的文件，这个文件是真实存储变量及其取值的文件。第二个是名为“`model.ckpt.meta`”的描述文件，在这个文件存储的是 `MetaGraphDef` 结构的对

¹² TensorFlow 0.12.0 版本之后，对于变量存储的定义升级为 V2 版本。此前的版本不会生成“`.index`”为后缀的文件，只有其他三个。

象经过二进制序列化后的内容。MetaGraphDef 结构由 Protocol buffer 定义，其中包含了整个计算图的描述、各个变量定义的声明、输入管道的形式，以及其他相关信息。meta 文件可以在没有计算图声明代码的情况载入模型，而若在申请时还有原始的 Python 程序代码，程序就已经可以重新构建计算图的基本信息，则加载只需要“model.ckpt”一个文件即可。第三个文件是以“model.ckpt.index”为名称的文件，存储了变量在 checkpoint 文件中的位置索引。最后一个名为“checkpoint”的文件，这个文件中存储了最新存档的文件路径。

模型存档有两种存取模式，除了上面示例展示的一次性存储之外，还有一种方式是通过引入迭代计数器的方式，按训练迭代轮次存储。使用这种方式时，需要在 save() 方法中指定当前迭代轮次，然后系统会自动生成带有测试的轮次和版本号的 checkpoint 文件。基本使用方式如下面代码所示：

```
with tf.Session() as sess:
    for step in range(max_step):
        # 执行迭代计算
        ...
        # 下面命令将生成以 'my-model.ckpt-???' 为文件名的 checkpoint
        saver.save(sess, 'my-model.ckpt', global_step=step)
```

由于每一轮迭代都会生成一组 checkpoint，在执行上万次迭代的训练过程中很可能把硬盘存储空间耗尽。为了防止这种情况发生，Saver 提供了几种有效的防范措施。第一种是设置 max_to_keep 参数，此参数指定存储操作以更迭的方式只保留最后几个版本的 checkpoint。默认情况下，只保留最后 5 个版本的存档。第二种是设置 keep_checkpoint_every_n_hours 参数，这种方式以时间为单位，每 n 个小时存储一个 checkpoint。该参数默认值是 10000，也就是每 1 万小时生成一个 checkpoint。

对于带有版本的 checkpoint 的加载有两种方法，一种是与之前的例子一样，直接指定名称前缀，加载一个特定版本的 checkpoint。另一种方式是利用名为“checkpoint”的文件，找到最新版本存档。

```
# 从 'checkpoint' 文件中读出最新存档的路径
ckpt = tf.train.get_checkpoint_state(ckpt_dir)
if ckpt and ckpt.model_checkpoint_path:
    # 找到合法存档，加载之
    saver.restore(sess, ckpt.model_checkpoint_path)
```

3.2.8 预测测试数据结果

作为最终的检验，训练完的模型需要对测试数据集进行预测，并提交至 Kaggle 平台验证预测准确率。

在前文中介绍了模型的搭建、训练和存档的过程，最后一步的预测相对之前就比较容易，简单来说就是加载测试数据后执行一遍正向传播计算即可。

```
# 读入测试数据集并完成预处理
testdata = pd.read_csv('data/test.csv')
testdata = testdata.fillna(0)
testdata['Sex'] = testdata['Sex'].apply(lambda s: 1 if s == 'male'
else 0)
X_test = testdata[['Sex', 'Age', 'Pclass', 'SibSp', 'Parch',
'Fare']]

# 开启 session 进行预测
with tf.Session() as sess:
    # 加载模型存档
    saver.restore(sess, 'model.ckpt')
    # 正向传播计算
    predictions = np.argmax(sess.run(y_pred, feed_dict={X: X_test}),
1)

# 构建提交结果的数据结构，并将结果存储为 csv 文件
submission = pd.DataFrame({
    "PassengerId": testdata["PassengerId"],
    "Survived": predictions
})
submission.to_csv("titanic-submission.csv", index=False)
```

从程序可以看到，执行正向传播计算，不需要代入标签数据，仅提供特征数据即可。

最终的结果按题目要求存储为 csv 格式。在 kaggle 提交后，可以看到平台给出的成绩：

Submission	Files	Public Score	Selected?
Mon, 21 Nov 2016 07:37:41 Edit description	titanic-submission.csv	0.72727	<input type="checkbox"/>
Mon, 21 Nov 2016 07:20:48 Edit description	titanic-submission.csv	0.47368	<input type="checkbox"/>

图 3-3 Titanic 提交结果展示

在此作者提交了两次，第一次尝试以随机分类为基线，即对于每个样本都随机给出 0 或 1 的判断，最终得到了 47% 的准确率，基本符合概率分布。第二次尝试是在训练逻辑回归分类器 10000 次迭代之后的结果，可以看到准确率有了明显的提升，达到了 72%，也就是对于超过三分之二的结果预测正确。考虑到 Titanic 问题本身的数据样本实在太少，逻辑回归模型又是最简单的模型，得到这样的准确率可以说已经是不错的结果了。

3.3 数据挖掘的技巧

在前一节我们使用 TensorFlow 实现了简单的逻辑回归分类器，并且取得了约超过七成的分类准确率。通过这个结果我们可以看到，虽然本题是一个简单的二分类问题，但是由于数据样本相对较少，而且问题本身受随机性的影响比较大，所以直接简单粗暴地使用分类器，结果还有很大的提高空间。在这一节中，我们提出一些优化方案，对提高预测准确率会有一定的帮助。

数据挖掘 (Data Mining) 是要从大量看似无序的数据中通过算法找到其中隐藏的信息和模式的过程，主要包括有监督的分类、预测和无监督的聚类、相关性分组等方法，比较经典的应用场景如垃圾邮件检测 (spam email detection)、“啤酒和尿不湿”故事等。

一般来说，解决数据挖掘类的问题没有一步到位的方法，都要经过不断尝试、反复分析优化的过程。大致来说，遵循下面这些“套路”：首先通过数据可视化，利用图形的方式更直观地感受数据，建立起足够的先验经验，然后利用特征工程方法筛选相关度最高的特征组合，最后将特征代入多种分类器进行试验，检验不同方法在同一

问题上的表现。

3.3.1 数据可视化

面对庞大的数据，仅仅依靠人脑很难建立起对数据的充分认识，使用工具以图形化的形式将数据展示出来，是必不可少的手段。

Titanic 问题的样本数据只有不到 1000 条，都有哪些可视化手段呢？下面这几种方式是 Kaggle 讨论组中给出的方案，可以作为参考。

图 3-4 是通过性别、客舱等级和是否有家人陪伴三个维度来考察幸存率的。从这样的图示中可以分析出如下几个结论：

1. 同等客舱的乘客中，女性乘客的幸存率普遍高于男性乘客。并且一等、二等客舱的女性乘客幸存率都非常高；
2. 一等、二等客舱的乘客中，有家人陪伴的会比独自乘船的存活率高；

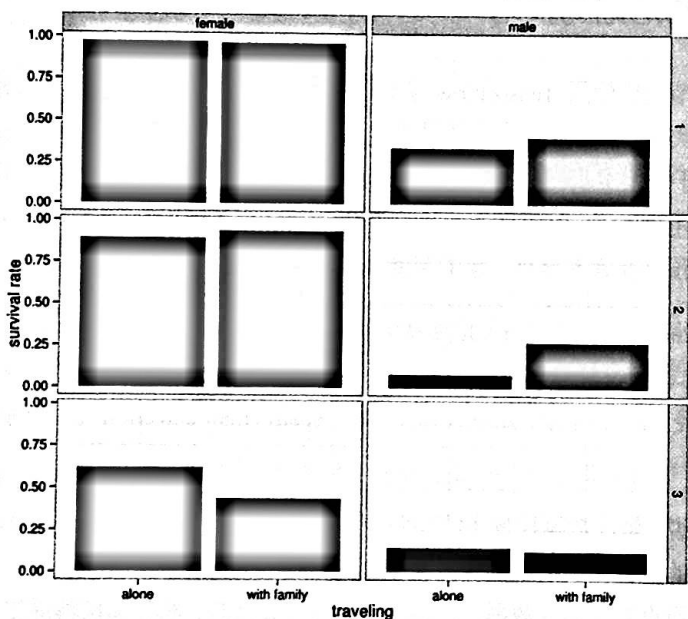


图 3-4 Titanic 数据中是否有家人陪伴的幸存率

3. 三等客舱中，独自乘船的乘客比有亲属的幸存率高。其中独自乘船的女性乘客有超过 50%幸存了下来。

4. 三等客舱的男性幸存概率只有 20%左右。

有了这样的分析结果，相信如果仅以创建规则的方式来判断乘客的存活率，也能有不错的表现。

另一种有趣的图形化方式是根据姓名，画出乘客的家族结构。如图 3-5 就展示了一等舱乘客的家庭关系。可以看到一等舱乘客基本都是家庭结伴而行的¹³。

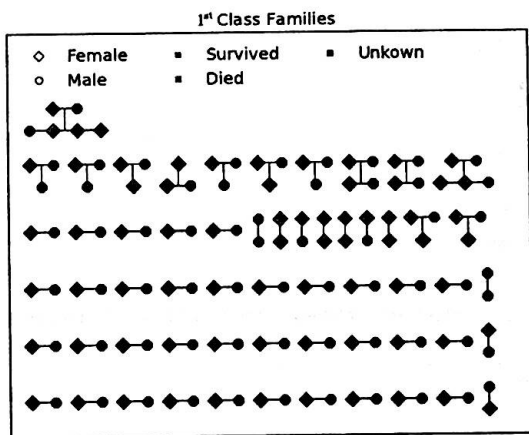


图 3-5 一等舱乘客的家族关系

再有一种比较有价值的分析是通过图形的方法，查看各个数据字段的重要程度。图 3-6 展示了对于随机森林算法而言，各个数据字段的重要程度，及重要度的标准差。可以看到，性别一项的重要度最高，其次是年龄。

通过这几种可视化分析的方法，从数据上验证了最初分析时提出的一些基础先验经验，比如女性和儿童的幸存率高，有身份的乘客幸存率高等，对于深入理解问题有极大帮助。

¹³ 全部乘客的家族结构示意图可查看 <https://www.kaggle.com/c/titanic/prospector#208>。

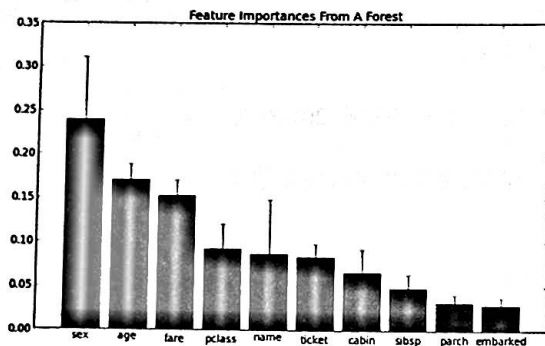


图 3-6 特征的重要程度分析

3.3.2 特征工程

对于传统机器学习领域来说,业界流行的说法是数据和特征决定了机器学习的上限,而模型和算法只是逼近这个上限的方法。所以若要追求更高的预测准确度,特征工程是必不可少的步骤。

特征工程是将原始数据的属性转换为特征的过程。以 Titanic 数据为例,属性是指数据的各个维度,比如乘客的性别、年龄、客舱等级,可以看作一组基础特征,而例如“是否为一等客舱的女性乘客”或“是否是 15 岁以下的儿童”之类的能够更好体现数据本质的表达方式,则是特征工程希望能得到的。在数据建模时,如果对原始数据的所有属性进行学习,由于噪声干扰较大,可能并不能很好地分析出数据的潜在趋势,拟合速度慢。而特征工程通过对数据进行重组,可以帮助算法模型减少噪声的干扰,得到更好的拟合效果。事实上,一组优秀的特征甚至能够帮助我们仅用简单的模型就达到很好的效果。

在上一节的实战中,我们只用到了 6 个变量字段,下面就介绍一下从其他字段中挖掘有用信息的方法。

数据清洗

数据清洗的目的是处理异常样本,如样本存在字段缺失,数据格式错误等问题,例如 Titanic 数据中乘客年龄字段就存在数据缺失的情况。针对这类样本有以下 4 种

常用的处理方式：

1. 直接丢弃整行样本数据，在样本数量足够的时候是优先的选择；
2. 当所有样本中某一字段缺失非常严重的时候，可以考虑丢弃整列字段；
3. 将缺失字段填充为默认值，如前面处理年龄字段就是补成默认值 0，相当于作为单独一种分类；
4. 将缺失字段填充为整列数据的平均值，这种做法相当于弱化了缺失项对其他特征的影响。

Titanic 数据就属于样本数量非常少的情况，训练数据只有 891 条，所以选择补齐字段是更合适的做法。

通过表 3-2 可以更清楚地看到，在样本数据中，年龄小于 15 岁的儿童有接近 60% 的幸存率，远超成人的 38%。而年龄缺失的人群中，幸存率更是低到不足三成。由此可以作出粗略的推断是，年龄字段缺失的样本应该属于成人，以年龄均值 29.6 岁作为标准补全数据即可。

表 3-2 小孩与成人的死亡率对比

	幸 存	死 亡	幸存率
小于 15 岁	49	34	59.0%
大于 15 岁	241	390	38.2%
年龄缺失	52	125	29.4%

用程序实现补全年龄数据可以利用 DataFrame 强大的编辑能力，两行代码即可完成。

```
mean_age = data["Age"].mean() # 29.69
data['Age'][data.Age.isnull()] = mean_age
```

数据预处理

数据预处理是通过数据进行离散化、标准化、归一化、数值变换等方法，使原本凌乱的数据更容易被算法处理。

一般来说,枚举类型或者取值范围是有限集合的字符串类型的字段,都会使用离散化方法转换成数值类型。方法也非常简单,只要构建一个枚举取值到数值的映射表即可。如前面提到过的 Titanic 数据中的性别字段就使用离散化的方法进行处理,从两个取值“male”和“female”分别转换为数字 1 和 0。另一个比较有趣的信息可以使用离散化处理,那就是乘客的头衔(Title)。乘客的头衔并不是原始数据中已经单独存在的属性,而是隐藏在乘客姓名中。乘客姓名一般都会带有“Mr.”、“Mrs.”、“Miss.”之类的称谓,甚至还有“Countess”、“Don”、“Dr.”等之类的尊称。称谓的不同代表了乘客身份上的差别,带有尊称头衔的乘客一定是贵族,乘坐一等客舱,有超高幸存的概率。因此,可以在解析姓名得到头衔后,标记该样本是否为贵族,作为一个备选特征。

```
def get_title(name):
    if pd.isnull(name):
        return 'Null'
    title_search = re.search('[A-Za-z]+\.', name)
    if title_search:
        return title_search.group(1).lower()
    else:
        return 'None'

titles = {'mr': 1,
          'mrs': 2, 'mme': 2,
          'ms': 3, 'miss': 3, 'mlle': 3,
          'don': 4, 'sir': 4, 'jonkheer': 4,
          'major': 4, 'col': 4, 'dr': 4, 'master': 4, 'capt': 4,
          'dona': 5, 'lady': 5, 'countess': 5,
          'rev': 7,}

data['Title'] = data['Name'].apply(lambda name:
    titles.get(get_title(name)))
data['Honor'] = data['Title'].apply(
    lambda title: 1 if title == 4 or title == 5 else 0)
```

对于定量的数值型特征而言,标准化是将整列数据按比例缩放,使之落在一个较小的区间内,比如将年龄从 [0, 100] 的区间等比缩小到 [0, 1] 区间中。还可以同时对定量特征进行二值化处理,比如以 15 岁为阈值,添加是否是儿童的标记特征。

Titanic 数据的一个突出的特点就是数据量实在比较少,在没办法取得更多数据的情况下,可以使用采样的方式随机增加一些数据。相比于直接增加训练迭代次数,随

机复制样本更不容易造成过拟合。

特征选择

在经过各种变换和处理后，我们将数据原本的几个属性，增加到了几十个特征组合。为了更好地提高计算效率，可能需要通过降维的方式，挑选对于分类结果区分度最大的一些特征组合。

下面列举的是一些最常用的方法：

1. 根据阈值过滤掉方差小的变量；
2. 通过计算变量与标签的相关系数，留下相关性高的特征；
3. 根据决策树或随机森林，选择重要程度高的特征；
4. 利用 PCA 等算法，对数据进行变换，选择区分度最高的特征组合。

特征选择的方法，大部分都在 `sklearn` 库中有对应的实现，在数据量比较小的情况下可以直接使用。

特征工程在传统机器学习研究中是公认的极其重要的部分。然而，手工的方式组合筛选特征，在像 Titanic 这种相对简单、特征维度较少的问题上还能做到不错的程度，但是在更复杂的更高维的问题上就会超出人类大脑能处理的极限。以图像识别问题为例，虽然对于人类来说，一个 2 岁的小朋友都能轻易而准确地在照片中识别出一只猫，但是很难用形式化的方法向计算机描述猫到底符合什么样的特征。所以，深度学习之所以强大，就是它可以通过大数据自动学习和提取出计算机才能看懂的特征集合。我们在后面的章节中会详细介绍面向图像和文字的深度学习算法。

3.3.3 多种算法模型

分类问题是机器学习中的基本问题，在经过了几十年的研究后，已经发展出了多种分类算法，如 Naive Bayes、线性回归、决策树、逻辑回归、KNN、SVM、神经网络、随机森林等，这些算法都能够应用于 Titanic 问题。

其实在选择算法模型的时候，并不一定是越复杂的模型越好。简单模型拟合能力弱，但收敛速度快，并且不易发生过拟合，而复杂的模型虽然有着更灵活的拟合能力，

但当数据量不足时更容易发生过拟合的情况。Titanic 题目就是典型的数据不足的例子。多次尝试和“用数据说话”，是最好的途径。

对于 Titanic 这一问题来说，由于数据量实在是非常少，而且本身存在比较大的随机性，所以在没有人为干预的情况下，一般能做到 79% 以上准确率，就已经是非常不错的成绩了。对于个别专家，使用机器学习算法最多也只能做到 85% 左右的预测准确率，可以以这一标准作为优化的最高目标¹⁴。

3.4 TensorBoard 可视化

从工程实践角度来说，往往逻辑越复杂的时候，代码写得就越长，其中产生 bug 的概率就越高，所以通常以“千行代码 bug 率”这一指标来衡量程序员的专业程度。一般使用 TensorFlow 的场景是做深度学习算法的开发和应用，而深度学习算法大多是非常复杂又难以理解的，往往涉及许多生涩难懂的数学公式，这就使得开发难度变得很高。不仅如此，机器学习算法普遍是以概率为基础的，存在一些不确定性，所以每一次训练的结果都可能存在一定的波动。当结果与预期存在一定偏差的时候，往往不能立刻定位是程序出错还是由于算法本身的随机性导致的，甚至有些错误代码还会引发正则化的效果，导致无法通过输出判断程序是否有 bug，这就进一步增大了程序调试的难度。值得庆幸的是，TensorFlow 系统的开发者们也希望解决这个问题。现在他们可以借助 TensorBoard 这一强大的可视化工具来帮助理解复杂的模型和检查实现中的错误。相比其他深度学习系统而言，TensorFlow 在这一方面做得最为先进和人性化。

目前 TensorBoard 可以展示几种数据：标量指标、图片、音频、计算图的有向图、参数变量的分布和直方图，还有最新添加的画出模型的计算图的图形，可以用曲线图的方式显示损失代价等量化指标的变化过程，还可以展示必要的图片和音频数据。

3.4.1 记录事件数据

TensorBoard 的工作方式是启动一个 Web 服务，该服务进程从 TensorFlow 程序执

¹⁴ Kaggle 上 Titanic 问题中，高于 85% 准确率的提交可以认为是经过其他人工方法矫正过的结果，属于作弊成绩。

行所得的事件日志文件（event files）中读取概要（summary）数据，然后将数据在网页中绘制成可视化的图表。概要数据主要包括以下几种类别：

1. 标量数据，如准确率、代价损失值，使用 `tf.summary.scalar` 加入记录算子；
2. 参数数据，如参数矩阵 `weights`、偏置矩阵 `bias`，一般使用 `tf.summary.histogram` 记录；
3. 图像数据，用 `tf.summary.image` 加入记录算子；
4. 音频数据，用 `tf.summary.audio` 加入记录算子；
5. 计算图结构，在定义 `tf.summary.FileWriter` 对象时自动记录。

与其他算子一样，记录概要的节点由于没有被任何计算节点所依赖，所以并不会自动执行，需要手动通过 `Session.run()` 接口触发。为了写起来更简便，`tf.summary.merge_all` 可以将所有概要操作合并成一个算子，其执行的结果是经过 `protocol buffer` 序列化后的 `tf.Summary` 对象。

完整的代码示例如下：

```
with tf.name_scope('input'):
    # create symbolic variables
    X = tf.placeholder(tf.float32, shape=[None, 6])
    y_true = tf.placeholder(tf.float32, shape=[None, 2])

with tf.name_scope('classifier'):
    # 分类器计算图
    weights = tf.Variable(tf.random_normal([6, 2]))
    bias = tf.Variable(tf.zeros([2]))
    y_pred = tf.nn.softmax(tf.matmul(X, weights) + bias)
    # 添加直方图参数概要记录算子
    tf.summary.histogram('weights', weights)
    tf.summary.histogram('bias', bias)

with tf.name_scope('cost'):
    cross_entropy = -tf.reduce_sum(y_true * tf.log(y_pred + 1e-10),
                                    reduction_indices=1)
    cost = tf.reduce_mean(cross_entropy)
    # 添加损失代价标量概要
```

```

tf.summary.scalar('loss', cost)

train_op = tf.train.GradientDescentOptimizer(0.001).minimize(cost)

with tf.name_scope('accuracy'):
    correct_pred = tf.equal(tf.argmax(y_true, 1), tf.argmax(y_pred,
1))
    acc_op = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
    # 添加准确率标量概要
tf.summary.scalar('accuracy', acc_op)

with tf.Session() as sess:
    # 创建概要写入操作
    # TensorBoard 可通过命令'tensorboard --logdir=./logs'来启动
    writer = tf.summary.FileWriter('./logs', sess.graph)
    # 方便起见, 合并所有概要算子
    merged = tf.summary.merge_all()

    for step in range(max_step):
        for i in range(batch):
            # 训练迭代
            ...
            summary, accuracy = sess.run([merged, acc_op],
                                         feed_dict={X: X_val, y_true:
y_val})
            writer.add_summary(summary, step)

```

生成的事件日志文件会是一个以“events.”开头的文件, 如前文所说, 里面的内容都是使用 protocol buffer 序列化之后的二进制数据, 只能通过 TensorBoard 打开。

3.4.2 启动 TensorBoard 服务

TensorBoard 是一个完整的 Python 应用, 通过命令行启动 Web。

```
$ tensorboard --logdir=path/to/log-directory
```

命令行的“--logdir”参数是 tf.summary.FileWriter 写入事件日志文件的目录路径。注意这里指定的是目录, 而不是文件。TensorBoard 在加载数据时会按顺序读取一个目录下的所有 TensorFlow 的事件文件, 这主要是因为当训练过程因为某些错误意外终止又被重新执行时, 会生成多个事件文件, 直接读取一个目录就能将多个文件的记录过程不间断地展示出来。另外, 如果指定的目录下有多个子目录的话, TensorBoard

会把它们当作多次执行的记录，可以并行展示出来。这一般是在多次运行，比较不同参数的效果时使用。

TensorBoard 命令行中其他常用的参数还有：

- `--port` 设置服务端口，默认端口为 6006。
- `--event_file` 指定某一个特定的事件日志文件。
- `--reload_interval` 服务后台重新加载数据的间隔，默认为每 120 秒。

本地服务启动后，可以在浏览器中打开 `http://localhost:6006/` 进行访问。图 3-7 和图 3-8 是 Tensorboard 的前端页面示例。TensorBoard 前端页面默认 120 秒自动刷新，所以如果是长时间训练的话，一直保持浏览器页面打开，就能够持续看到程序执行的进程。

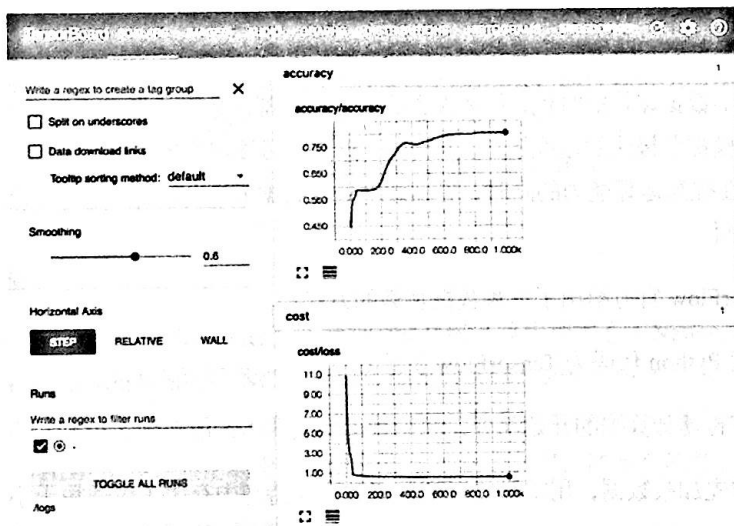


图 3-7 TensorBoard 标量可视化

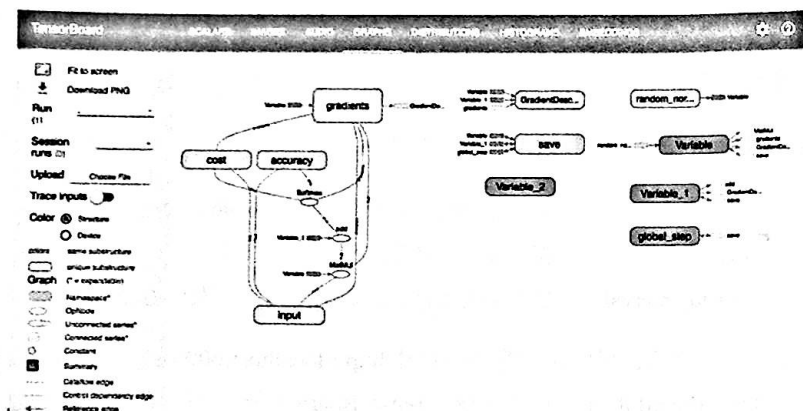


图 3-8 TensorBoard 计算图可视化

3.5 数据读取

在当前数据爆炸的时代，几乎人类的一切行为都以数据的形式被记录下来，各种数据的规模都以极快的速度增长。同样的，机器学习领域的各种数据集的体积也是越来越大。在提高运算能力的同时，更高效地处理数据 I/O 也是提高整体性能非常重要的一个方面。

TensorFlow 官方给出了三种数据加载的方式：

1. 用 Python 代码为 TensorFlow 供给数据；
2. 在构建计算图的开始部分，利用管道从文件中读取数据；
3. 预先加载数据，用常量或者变量将数据保持在内存中（仅适用于小数据）。

对于体积较小的数据，直接预先加载全部样本到内存，然后分 batch 输入网络训练是最方便的选择。而对于几百 GB 甚至 TB 级别的大数据而言，加载到内存就不太现实了，一方面对内存消耗巨大不一定放得下，另一方面分块读取的话频繁 I/O 也会造成执行效率大打折扣。因此，选择合适的输入数据格式，也是一个需要着重关注的方面。

3.5.1 数据文件格式

对于可以直接加载到内存或显存中的比较少量的数据（一般为 1GB 以内的级别），数据读入一般在 1 分钟内就可以完成，也不会耗尽内存或显存，因此不会对整个训练过程造成瓶颈。对于这种量级的数据而言，除了 csv 格式以外，还有一些高性能的格式文件类型可以选择，例如 npy 和 npz 格式、pkl 格式和 hdf 格式：

npz 和 npy 格式

NumPy 库作为高性能计算最常用的库，自然有处理数据 I/O 所需要的方法。numpy.save() 方法就可以将数组存储为扩展名为“.npz”的二进制文件。用 numpy.load() 读出时，它可以自动处理元素的类型和数组维度等信息。npz 文件是存储多个数组数据的文件格式，其内部实际是将多个 npy 文件归档。

pkl 格式

pickle 是 Python 内置的数据序列化和反序列化模块，通过该模块可以将 Python 对象持久化成 pkl 格式的文件。cPickle 是与 pickle 在功能和用法上几乎相同的包，但由于是使用 C 语言编写的，所以在性能上比 pickle 高出 1000 倍。一般若是使用 pkl 格式，则用 cPickle 库进行操作。

hdf 格式

HDF（Hierarchical Data File）是美国国家高级计算应用中心（National Center for Supercomputing Application, NCSA）为了满足各种领域研究需求而研制的一种能高效存储和分发科学数据的新型数据格式，HDF5 是其系列中最新，也是目前最常用的一种格式。HDF 文件是通用的自描述的数据文件格式，可以跨平台高效读写。

3.5.2 TFRecord

对于大数据，TensorFlow 推荐使用自家的 TFRecord 文件。TFRecord 文件同样是以二进制进行存储数据的，适合以串行的方式读取大批量数据。TFRecord 内部的格式虽然略为复杂不易理解，但是它能更好地利用内存，更方便地复制和移动，更符合 TensorFlow 执行引擎的处理方式。

普通的数据很容易转换成 TFRecord 格式的文件。只需要写一个小程序，将每一

条样本组装成 protocol buffer 定义的 Example 结构的对象，序列化成字符串，再由 `tf.python_io.TFRecordWriter` 写入文件即可。这里给出示例将 Titanic 数据转换成 TFRecord 格式，介绍基本的 TFRecord 文件读/写操作及其使用。关于大数据的多线程写入操作、关键函数解析、变长序列样本 `SequenceExample` 的处理和序列样本 batch 包的构建等相关函数概念将在第 6 章解释，并给出完整的代码示例。Titanic 数据转换示例如下：

```
# 将 train.csv 转换为 train.tfrecords
def transform_to_tfrecord():
    data = pd.read_csv('data/train.csv')
    tfrecord_file = 'train.tfrecords'

    def int_feature(value):
        return tf.train.Feature(
            int64_list=tf.train.Int64List(value=[value]))

    def float_feature(value):
        return tf.train.Feature(
            float_list=tf.train.FloatList(value=[value]))

    writer = tf.python_io.TFRecordWriter(tfrecord_file)
    for i in range(len(data)):
        features = tf.train.Features(feature={
            'Age': float_feature(data['Age'][i]),
            'Survived': int_feature(data['Survived'][i]),
            'Pclass': int_feature(data['Pclass'][i]),
            'Parch': int_feature(data['Parch'][i]),
            'SibSp': int_feature(data['SibSp'][i]),
            'Sex': int_feature(1 if data['Sex'][i] == 'male' else 0),
            'Fare': float_feature(data['Fare'][i])
        })
        example = tf.train.Example(features=features)
        writer.write(example.SerializeToString())
    writer.close()
```

从 TFRecord 文件中读出数据，使用 `TFRecordReader`。`TFRecordReader` 是一个算子，因此 TensorFlow 能够记住 `tfrecords` 文件读取的位置，并且始终能返回下一条记录。

`tf.train.string_input_producer` 方法用于定义 TFRecord 文件作为模型结构的输入部

分。该函数输入文件名列表在 Session 运行时产生文件路径字符串循环队列。

根据产生的文件名，TFRecordReader.read 方法打开文件，再由 tf.parse_single_example 方法解析成一条可用的数据。tf.train.shuffle_batch 可以设置内存读取样本的上限与上限训练 batch 批次的大小等参数，用于定义产生随机生成的 batch 训练数据包。

在 Session 的运行中，tf.train.shuffle_batch 函数生成 batch 数据包的过程是作为线程独立运行的。数据输入线程的挂起和运行时机由 batch 数据的生成函数控制。本例中的 tf.train.shuffle_batch 函数指定内存保存样本数量的上限 capacity 和下限 min_after_dequeue。当内存中保存的样本数量大于上限 capacity 时，数据输入线程挂起。反之，当样本数量小于 min_after_dequeue 时，训练程序挂起。函数 start_queue_runners 开启对应运行会话 Session 的所有线程队列并返回线程句柄。Coordinator 类对象负责实现数据输入线程的同步。当 string_input_producer 函数产生无限循环队列时，应取消数据输入与训练程序的线程同步。读入的示例程序如下：

```
import tensorflow as tf

def read_and_decode(train_files, num_threads=2, num_epochs=100,
                    batch_size=10, min_after_dequeue=10):
    # read data from trainFile with TFRecord format
    reader = tf.TFRecordReader()
    filename_queue = tf.train.string_input_producer(
        train_files,
        num_epochs=num_epochs)
    _, serialized_example = reader.read(filename_queue)
    featuresdict = tf.parse_single_example(
        serialized_example,
        features={
            'Survived': tf.FixedLenFeature([], tf.int64),
            'Pclass': tf.FixedLenFeature([], tf.int64),
            'Parch': tf.FixedLenFeature([], tf.int64),
            'SibSp': tf.FixedLenFeature([], tf.int64),
            'Sex': tf.FixedLenFeature([], tf.int64),
            'Age': tf.FixedLenFeature([], tf.float32),
            'Fare': tf.FixedLenFeature([], tf.float32)})

    # decode features to same format of float32
    labels = featuresdict.pop('Survived')
```

```

features = [tf.cast(value, tf.float32)
             for value in featuresdict.values()]

# get data with shuffle batch and return
features, labels = tf.train.shuffle_batch(
    [features, labels],
    batch_size=batch_size,
    num_threads=num_threads,
    capacity=min_after_dequeue + 3 * batch_size,
    min_after_dequeue=min_after_dequeue)
return features, labels

def train_with_queuerunner():
    x, y = read_and_decode(['train.tfrecords'])

    with tf.Session() as sess:
        tf.group(tf.global_variables_initializer(),
                  tf.local_variables_initializer()).run()

        coord = tf.train.Coordinator()
        threads = tf.train.start_queue_runners(sess=sess, coord=
coord)

        try:
            step = 0
            while not coord.should_stop():
                # Run training steps or whatever
                features, labels = sess.run([x, y])
                if step % 100 == 0:
                    print('step %d:' % step, labels)
                    step += 1
            except tf.errors.OutOfRangeError:
                print('Done training -- epoch limit reached')
            finally:
                # When done, ask the threads to stop.
                coord.request_stop()
                # Wait for threads to finish.
                coord.join(threads)

if __name__ == '__main__':
    train_with_queuerunner()

```


3.6 SkFlow、TFLearn 与 TF-Slim

TensorFlow 在接口抽象方面拥有非常合理的设计,但从工程化的角度而言,自然是希望能够用更少的代码实现更复杂的逻辑。Scikit-Learn 是机器学习方面的标杆系统,其接口简洁易用,只需要很少的几行代码就可以实现一个分类器模型。因此,将 Scikit-Learn 的接口嫁接到 TensorFlow 上就成为了一件非常有吸引力的事。

SkFlow¹⁵是 TensorFlow 官方推出的仿照 Scikit-Learn 设计的高级 API。其中对多种常用的分类回归模型进行了封装,使得实现一个分类器仅需要几行代码即可完成。

```
import tensorflow.contrib.learn as skflow
from sklearn import metrics

feature_cols = skflow.infer_real_valued_columns_from_input(X_train)
classifier = skflow.LinearClassifier(feature_columns=feature_cols,
                                     n_classes=2)
classifier.fit(X_train, Y_train, steps=200)
accuracy = metrics.accuracy_score(Y_val,
classifier.predict(X_val))
print("Accuracy: %f" % accuracy)
```

另一个对 TensorFlow 做了高级封装的项目是 TFLearn。这是一个完全由开源社区贡献完成的项目,在 GitHub 上获得了很高的 Star 数,是一个非常优秀而且应用广泛的项目。

从命名即可看出,TFLearn 与 SkFlow 一样也是仿照 Scikit-Learn 来做的接口设计,同样对多种常用分类回归模型进行了封装,也同样是 fit 一下即可完成整个训练过程。

用 TFLearn 实现 Titanic 题目的逻辑回归,代码是 TensorFlow 原生实现的一半:

```
import os

import numpy as np
import pandas as pd
import tensorflow as tf
import tflearn
```

¹⁵ 现已更名为 TF Learn,为了避免与另一个项目混淆,本书中仍然将其称为 SkFlow。现已与 TensorFlow 项目完全集成,成为 TensorFlow 中的一个包 `tf.contrib.learn`。

```

# 读取训练数据
train_data = get_train_data()
X = train_data[['Sex', 'Age', 'Pclass', 'SibSp', 'Parch', 'Fare',
'Child',
                'EmbarkedF', 'DeckF', 'TitleF', 'Honor']].as_matrix()
Y = train_data[['Deceased', 'Survived']].as_matrix()

# arguments that can be set in command line
tf.app.flags.DEFINE_integer('epochs', 10, 'Training epochs')

# 创建存档目录
ckpt_dir = './ckpt_dir'
if not os.path.exists(ckpt_dir):
    os.makedirs(ckpt_dir)

# 定义分类模型
n_features = X.shape[1]
input = tflearn.input_data([None, n_features])
y_pred = tflearn.layers.fully_connected(network, 2, activation='softmax')
net = tflearn.regression(y_pred)
model = tflearn.DNN(net)

# 读取模型存档
if os.path.isfile(os.path.join(ckpt_dir, 'model.ckpt')):
    model.load(os.path.join(ckpt_dir, 'model.ckpt'))
# 训练
model.fit(X, Y, validation_set=0.1, n_epoch=tf.app.flags.FLAGS)
# 存储模型参数
model.save(os.path.join(ckpt_dir, 'model.ckpt'))
# 查看模型在训练集上的准确率
metric = model.evaluate(X, Y)
print('Accuracy on train set: %.9f' % metric[0])

# 读取测试数据, 并进行预测
test_data = get_test_data()
X = test_data[['Sex', 'Age', 'Pclass', 'SibSp', 'Parch', 'Fare',
'Child',
                'EmbarkedF', 'DeckF', 'TitleF', 'Honor']].as_matrix()
predictions = np.argmax(model.predict(X), 1)
submission = pd.DataFrame({

```

```

    "PassengerId": test_data["PassengerId"],
    "Survived": predictions
})

submission.to_csv("titanic-submission.csv", index=False)

```

可以看到，使用 SkFlow 和 TFLearn 这种高级接口可以大幅度地节省代码量，在加快开发速度的同时，也意味着降低 bug 率，并且让代码的可读性更高。

TensorFlow-Slim（简称 TF-Slim）也是 TensorFlow 官方开放的一个轻量级的高级接口库，使用这个库可以让复杂模型的定义、训练、评估都变得简单。TF-Slim 在图像模型方面有较大的优势，它包含了很多新的层（如 Atrous 卷积层）和新的评估标准（如 mAP、IoU），还内置了包括 AlexNet、inception、VGGNet、OverFeat、ResNets 在内的各种经典图像识别模型。

对于深度学习模型的开发来说，使用 SkFlow、TFLearn 或 Slim 相对于使用原生接口编程都有比较大的优势，不但对卷积、递归神经网络等神经元层进行了高度封装，而且还能达到自动处理各种事件日志、显示训练进度、简化模型存档等功效，往往能达到一行代码胜千言的理想状态，极大地提高开发效率。

3.7 小结

通过这一章的介绍可以看到，TensorFlow 的安装和使用都十分简单，对于机器学习算法开发有着极大的帮助，让机器学习不再是一件只有专家才能做的事。同时，TensorBoard、TFLearn 等一众辅助工具和扩展项目还能进一步降低 TensorFlow 的上手门槛，为大规模深度学习应用奠定了基础。

3.8 参考资料

- [1] TensorFlow Tutorial: https://www.tensorflow.org/versions/r0.11/how_tos/
- [2] 周志华，《机器学习》。北京：清华大学出版社，2016。
- [3] NumPy 项目官方网站；<http://www.numpy.org/>

- [4] Pandas 项目官方网站: <http://pandas.pydata.org/>
- [5] Matplotlib 项目官方网站: <http://matplotlib.org/>
- [6] PIL 项目官方网站: <http://www.pythonware.com/products/pil/>
- [7] Jupyter 项目官方网站: <http://jupyter.org/>
- [8] Scikit-learn 项目官方网站: <http://scikit-learn.org/>
- [9] OpenCV 项目官方网站: <http://opencv.org/>
- [10] Kaggle 官方网站: <https://www.kaggle.com/>
- [11] TFLearn 官方文档: <https://tflearn.org/>
- [12] Python 的 Pickle 包: <https://docs.python.org/2/library/pickle.html>
- [13] <https://www.zhihu.com/question/28641663/answer/41653367>

4

CNN “看懂” 世界

第3章，通过 Titanic 挑战问题，介绍了 TensorFlow 在传统机器学习中的基本使用方式。如前文所说，特征工程在传统机器学习中占有至关重要的地位。由于特征工程依赖于人类对问题本身的理解，所以很难将程序推广到其他类似的实际应用中。例如，在图像分类问题中，传统的机器学习对于不同颜色和不同形状的目标需要使用不同的特征提取算法获取特征。深度卷积神经网络为图像分类提供了统一的解决方案。

对于人类而言，视觉是认识世界最重要的渠道，大脑每天要处理的信息中，通过视觉感官接收到的信息占了 80% 以上。而对于计算机来说，虽然也可以通过镜头“看到”所有的画面，但是“看懂”画面里的内容则并不是一件容易的事情。一张图片中包含的语义信息错综复杂，但在计算机看来则只是一个个零散而独立的像素点。如何以计算机的语言表达像素与像素之间的语义关系，是最大的挑战。

卷积神经网络 (Convolutional Neural Networks, 简称 CNNs)，由纽约大学的 Yann LeCun 教授于 1989 年发明，是一种专门为处理高维网格型数据（也就是张量 Tensor）而设计的神经网络。卷积神经网络最擅长处理图像数据，例如用二维矩阵表示的灰度图像，三维数组（高、宽、RGB 通道）表示的彩色图片等，也在该领域取得了令世界瞩目的成绩。在 2012 年 ILSVRC 图像识别竞赛中，Alex Krizhevsky 基于卷积神经网络设计的分类模型 AlexNet 大放异彩，以压倒性优势赢得了当年的冠军，体现了

CNNs 在图像识别问题上的强大能力，同时也体现了深度学习的巨大潜力，瞬间使 CNNs 和深度学习成为了科学家们追捧的热门研究领域。

在本章中，首先会简单介绍卷积神经网络的基本原理，随后介绍各种经典的深度卷积神经网络模型及 TensorFlow 实现，最后，通过图像风格化应用开启新世界的大门，看看 CNNs 除了物体识别外还有什么本领。

4.1 图像识别的难题

机器学习的核心问题是分类和回归，对于图像来说也不例外。著名的 ILSVRC 图像识别竞赛中最基础的一项就是要求参赛者对海量的图像进行识别并分类。在识别分类的基础上，可以扩展实现目标定位和多目标检测。

ILSVRC (ImageNet Large Scale Visual Recognition Challenge) 是近年来机器视觉领域最受追捧也是最具权威的学术竞赛之一。每年全世界最顶尖的科学家和企业都会参与到这个盛会中，利用最前沿、最先进的算法来解决图像识别方面的难题，并不断刷新各种挑战的记录，代表了图像领域的最高水平。几乎每一年冠军队伍提出的新模型，都是来年 CVPR 会议上的大热门。

ImageNet 数据集是 ILSVRC 竞赛中使用的数据集，由斯坦福大学著名的华人教授李飞飞主导，其中包含了超过 1400 万张全尺寸的有标记图片。ILSVRC 比赛会每年从 ImageNet 数据集中抽出部分样本，以 2012 年为例，比赛的训练集包含 1281167 张图片，验证集包含 50000 张图片，测试集为 100000 张图片。

ILSVRC 竞赛的比赛项目实际上是图像识别和计算机视觉领域中最困难、应用范围最广、最需要解决的问题。它包括如下几个问题：

1. 图像分类与目标定位 (CLS-LOC)

图像分类与目标定位最初是两个独立的项目。图像分类的任务是要判断图片中的物体在 1000 个分类中所属的类别，主要采用 top-5 错误率的评估方式，即对于每张图给出 5 次猜测结果，只要 5 次中有一次命中真实类别就算正确分类，最后统计完全没有命中的错误率。2012 年之前，图像分类最好的成绩是 26% 的错误率，2012 年

AlexNet 的出现降低了 10 个百分点，错误率降到 16%，而到了 2016 年，由公安部第三研究所选派的“搜神”（Trimps-Soushen）代表队在這一项目中获得冠军，将成绩提高到仅有 2.9% 的错误率。

目标定位项目是要在分类正确的基础上，从图片中标识出目标物体所在的位置，用方框框定，以错误率作为评判标准。其难度在于，图像分类问题可以有 5 次尝试机会，而在目标定位问题上，每一次都需要框定得非常准确。在这一项目上，2015 年 ResNet 贡献了巨大的提高，从上一年的最好成绩 25% 的错误率，提高到了 9%。2016 年该项目的冠军同样是公安部三所的 Trimps-Soushen 代表队，错误率仅为 7.7%。

2. 目标检测 (DET)

目标检测是在定位的基础上更进一步，在图片中同时检测并定位多个类别的物体。具体来说，是要在每一张测试图片中找到属于 200 个目标类别中的所有物体，如人、勺子、水杯等。最终的评判方式是看模型在每一个单独类别中的识别准确率，在多数类别中都获得最高准确率的队伍获胜。平均检出率 mean AP (mean Average Precision) 是这一项上的重要指标，一般来说，平均检出率最高的队伍也会在多数的独立类别中获胜，2016 年这一成绩达到了 66.2%。

3. 视频目标检测 (VID)

视频目标检测与图片目标检测任务类似，是要检测出视频每一帧中包含的多个类别的物体。要检测的目标物体有 30 个类别，是目标检测 200 个类别的子集。此项问题最大的难度在于要求算法的检测效率非常高。评判方式与目标检测相同，在独立类别识别最准确的队伍获胜。2016 年南京信息工程大学队伍在这一项目上获得了冠军，他们提供的两个模型分别在 10 个类别中胜出，并且达到了平均检出率超过 80% 的好成绩。

4. 场景分类 (Scene)

场景分类是要识别图片中的场景，比如森林、剧场、会议室、商店等。也可以说，场景分类要识别图像中的背景。这个项目由 MIT Places 团队组织，使用 Places2 数据集，其中包括 400 多个场景的超过 1000 万张图片。评判标准与图像分类相同，5 次猜测中有一次命中即可，最后统计错误率。2016 年最佳成绩的错误率仅为 9%。

场景分类问题中还有一个子问题是场景分割，是要将图片划分成不同的区域，比如天空、道路、人、桌子等。该项目由 MIT CSAIL 视觉组组织，使用 ADE20K 数据集，包含 2 万多张图片，150 个标注类别，例如天空、玻璃、人、车、床等。这个项目会同时评估像素级准确率和分类 IoU (Intersection of Union)。

除了 ILSVRC 竞赛所列出的题目之外，图像领域的深度学习探索还有许多有趣的问题，比如看图说话 (Image Captioning) 是为给定的图片配上一段说明文字，图像纹理提取的是图像中的纹理特征。更有一些无监督学习的探索，比如 Google 让计算机自己从无标注的图片中学会了识别猫的样子。

总之，可以说图像识别是一项基础研究，应用场景非常丰富，所要面临的问题也非常有挑战，仍然需要不懈地研究下去。

4.2 CNNs 的基本原理

在卷积神经网络出现之前，也有人尝试使用人工神经网络 (ANNs) 解决图像分类问题，但是一直没有取得很好的效果。

如果了解传统神经网络就会知道，作为神经元的感知机 (perceptron) 模型的结构非常简单，用数学方法表示就是矩阵相乘后进行非线性变换，即

$$f(x) = \text{sign}(xW + b)$$

在这里面，权值矩阵 W 实际表示了输入节点与输出节点之间的影响关系。如果输入节点与输出节点之间的权值参数的绝对值越大，就代表两个节点的相关性越高。在全连接结构中，每一个节点的输入都是上一层的所有输出，当隐层节点数目较多时，整个网络中的参数数目会变得非常巨大。以图片来说，一般 jpg 格式的图片在计算机中以三维矩阵的形式存储。假设要处理一张分辨率为 1000×1000 的灰度图，则一共有 $1000 \times 1000 \times 3 = 3000000$ 个 uint8 类型的整数值。若以每一个值都是一个特征维度，想要经过一层感知机的变换后输出维度不变，就需要一个 $1000 \times 1000 \times 1000 \times 1000 = 10^{12}$ 个元素的参数矩阵，如图 4-1 所示。在这种情况下，不管是权值矩阵占用的存储空间，还是模型的计算量都是海量的。由此导致的问题是，参数过多则收敛速度慢，需要更多的数据和更多的训练迭代，但巨大的计算量意味着不可能在

有限的时间内完成所需要的计算。

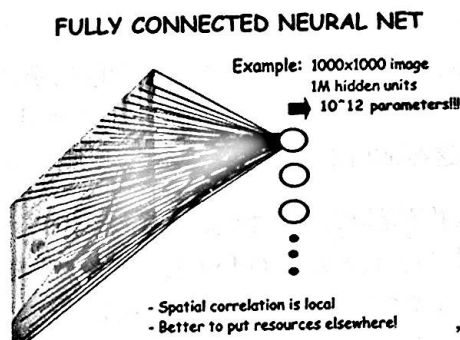


图 4-1 全连接神经网络处理图像会导致参数爆炸

从另一个角度来说，其实从语义上来理解也同样存在问题：全连接的网络结构在处理每一个像素时，其相邻像素与距离很远的像素都是无差别地对待的，并没有考虑图像内容的空间结构。但一般来说图像的语义并不以像素为单位，而是各种连续的线条、形状或色块，要让全连接网络模型通过数据从零开始学习其中的模式规律，自然是非常困难的，需要极大的代价，倒不如直接设计一种结构代入人类的先验经验来处理这种空间结构。

由此，卷积神经网络应运而生。卷积神经网络是指：至少有一层计算为卷积操作的神经网络。卷积操作是其中的核心，它与全连接结构最大的不同，就是它充分利用了图片中相邻区域的信息，通过稀疏连接和共享权值的方式大大减少参数矩阵的规模，从而减少计算量，也大大地提高了收敛速度。那什么是卷积呢？具体又是如何减少参数规模的呢？

4.2.1 卷积的数学意义

卷积原本是一种积分变换的数学方法，是通过两个函数 f 和 g 生成第三个函数的算子。可以通过下面这个例子来更好地理解卷积的概念。

想象这样一个场景：在一个冬日慵懒的午后，你正窝在沙发上拿着手机刷微博、刷微信、刷知乎、刷 twitter（咦？twitter？），突然屏幕上弹出一个白色弹窗，“电量不足 20%，请充电”，斜眼一瞟，果然电量显示已经飘红，惊慌中你匆忙从沙发上弹起来，找到充电线插上，嗯，安心了，继续刷刷刷……那么现在问题来了，从一边使

用手机一边给手机充电的时候开始算起, 请问在某一时刻 t 的手机电池的充电电量是多少?

如果假设手机的耗电速度和充电速度都是恒定值的话, 那这个问题就变成了我们都熟知并令许多人深恶痛绝的蓄水池问题, 即一个水管进水另一个水管放水, 求何时可以储满或放空水池。这种简单问题小学生就能搞定。但是实际情况并没有那么理想。

首先, 电量的输入并不是恒定的, 由于我们现在使用的基本上都是交流电, 所以可以认为充电的电量是与正弦函数 $\sin()$ 相关的函数, 我们将 t 时刻的输入表示为 $x(t)$, x 和 t 的取值都是实数域, 因为时间和正弦函数是连续的。其次, 电量的消耗也不是恒定值, 因为手机在静置待机状态下和正常使用状态下的用电量肯定不同。在这里对于电量消耗可以换一种理解, 认为消耗的电量其实是对之前所充入电量的衰减, 也就是说充入电量的功效时刻在经历着衰减。那么可以将电量的衰减系数表示为 $w(a)$, 其中 a 是测量的时长。将这两个函数结合起来, 就得到了测量某时刻 t 的手机电量的表示:

$$s(t) = \int x(a) w(t-a) da$$

这一操作就叫做卷积, 它也被通常被简化地表示成星号的形式:

$$s(t) = (x * w)(t)$$

卷积操作实际上是对输入函数的每一个位置进行加权累加。 $x(t)$ 是输入信号, 衰减系数 $w(t-a)$ 是系统对信号的响应, 卷积 $(x * w)(t)$ 就是在时刻 t 对系统观察的结果, 是所有信号经过系统的处理后的结果的叠加。

从实际数据来说, 由于不可能真的做到实时测量, 所以一般是将时间离散化后, 按固定的时间间隔进行测量, 所以数据不会是连续的。可以假设只考虑整数时刻, 即 t 为整数的情况, 则离散后的卷积可以表示为:

$$s(t) = (x \times w)(t) = \sum x(a)w(t-a)$$

离散化后的公式就稍微容易理解一些。可以理解为, 单位时间输入的电量会随着使用被逐渐消耗, 在时刻 t 的手机电量, 就是之前各个单位时间充入电量中尚未衰减掉的剩余值的累加结果。

4.2.2 卷积滤波

卷积其实是图像处理中一种常用的线性滤波方法，使用卷积可以达到图像降噪、锐化等多种滤波效果。

图像的卷积的计算过程并不复杂，对于一张二维的图片 I 和一个二维的卷积核滤波矩阵 K ，卷积操作可以表示成：

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n) K(i - m, j - n)$$

简单来说，就是对于图片中的每一个像素点，计算它的邻域像素和滤波器矩阵的对应位置元素的乘积，然后将所有乘积累加，作为该像素位置的输出值¹⁶。卷积核依次滑过图片中的每一个像素位置，就可以输出一张分辨率从来不变的新图片。从数学上说，就是求两个矩阵的滑动点积或者滑动内积，如图 4-2 所示。

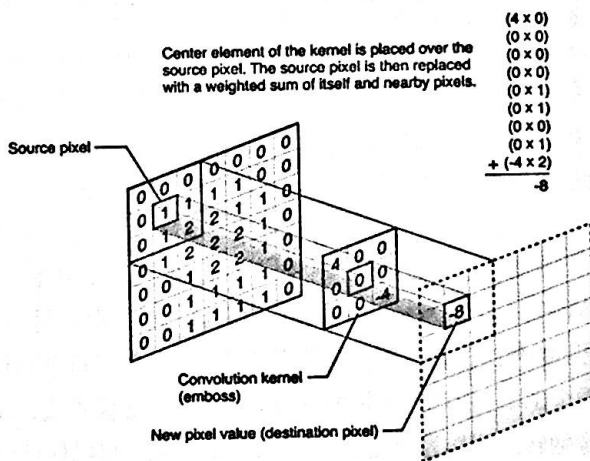


图 4-2 卷积操作的过程示例

可以看到，所谓的卷积核，实际就是一个权值矩阵，表示如何处理单个像素与其邻域像素之间的关系。卷积核中各个元素的相对差值越小，相当于每个像素都与周围

¹⁶ 卷积与协相关 (cross-correlation) 是信号处理中两种常用的操作，在计算上非常相似，严格来说，卷积需要先对卷积核矩阵进行翻转。然而对于机器学习来说并不重要，因为算法会调整每一个参数值，并最终把合适的值放在合适的位置。

像素取了个平均值,就越有模糊降噪的效果。而卷积核元素的差值越大,就拉大了每个像素与周围像素的差距,也就越能提取边缘,或者达到锐化的效果。

在实际应用中,卷积核矩阵的尺寸一般都比较小,主要原因是卷积核尺寸与计算量成正比,卷积核尺寸增大会使得计算量成倍增加。同时,一般卷积核矩阵为正方形,并且边长为奇数,比如 3×3 、 5×5 或者 7×7 ,这样才能保证有且只有一个中心,输出的时候可以与原图的像素有所对应。当然这也并不是必须的,非对称卷积核同样可以计算。

卷积核矩阵中各个元素的取值,除了影响输出效果以外,还会影响输出图片的亮度。若卷积核内所有元素的累加和等于 1,那就基本上保持了与原图相同的亮度。若大于 1,则输出图会变亮,若小于 1,输出图的亮度就会变暗。当卷积核中元素累加和为 0 时,输出的图片亮度会非常低,但并不是全黑,而是大部分面积都是黑色,仅有部分图案的边缘还有一些亮度,这样的卷积核就可以用于边缘提取。

卷积操作中,对于图像边缘像素的处理是值得注意的地方。在卷积核矩阵滑过图片每一个像素的过程中,当遇到图像边缘的时候,例如图像顶部的像素,如果把卷积核的中心对应到图像像素,那么卷积核上面几个位置就没有与之对应的像素了,这应该如何计算呢?此时有两类处理方式。一类叫做“Valid Padding”,是直接忽略这种无法计算的边缘像素,只计算那些能计算的部分。这种做法的输出图片会比原图尺寸要小。比如使用 3×3 的卷积核来卷积一张 800×600 的图片,则输出图片的尺寸是 798×598 ,也就是上下左右每一个方向都减少了一个像素。另一类叫做“Same Padding”,是对原图的外围进行无限填充使得边缘像素能够进行合法计算。在这种方法中,可以将外围全部填充为 0,或是填充为最临近的边缘像素值,或是认为图片是无限循环的,镜像翻转图片作为填充值。“Same Padding”可以保证输出图片的尺寸与原图保持不变,这是常用的方法。

在图像处理中,通常会用到一些经典的卷积滤波器,如低通滤波器、高通滤波器、高斯滤波器,这些滤波器会产生不同的效果,图 4-3 展示了一些具体的例子。

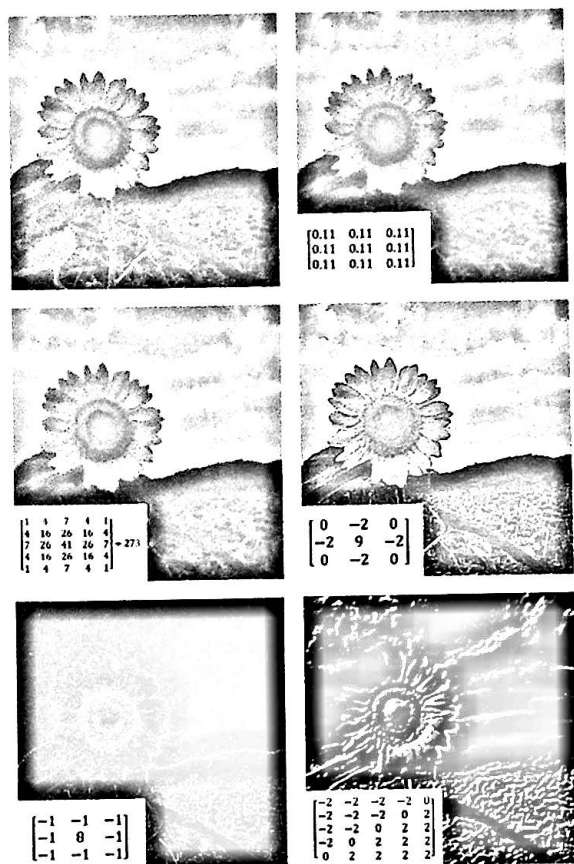


图 4-3 各种卷积滤波器的效果。依次为：原图、低通滤波器（Low Pass Filter）、高斯滤波器（Gaussian Filter）、锐化滤波器（Sharpeness Filter）、边缘检测（Edge Detection）、浮雕滤波器（Embossing Filter）

各种效果可以使用 opencv 轻易实现出来：

```
import cv2
import numpy as np

# 读入原图
image = cv2.imread("flower.jpg")
cv2.imshow('original', image)

# 低通滤波
```

```

kernel = np.array([[.11, .11, .11],
                   [.11, .11, .11],
                   [.11, .11, .11]])
rect = cv2.filter2D(image, -1, kernel)
cv2.imwrite('rect.jpg', rect)

# 高斯滤波
kernel = np.array([[1, 4, 7, 4, 1],
                   [4, 16, 26, 16, 4],
                   [7, 26, 41, 26, 7],
                   [4, 16, 26, 16, 4],
                   [1, 4, 7, 4, 1]]) / 273.0
gaussian = cv2.filter2D(image, -1, kernel)
cv2.imwrite('gaussian.jpg', gaussian)

# 锐化
kernel = np.array([[0, -2, 0],
                   [-2, 9, -2],
                   [0, -2, 0]])
sharpen = cv2.filter2D(image, -1, kernel)
cv2.imwrite('sharpen.jpg', sharpen)

# 边缘检测
kernel = np.array([[ -1, -1, -1],
                   [-1, 8, -1],
                   [-1, -1, -1]])
edges = cv2.filter2D(image, -1, kernel)
cv2.imwrite('edges.jpg', edges)

# 浮雕
kernel = np.array([[ -2, -2, -2, -2, 0],
                   [-2, -2, -2, 0, 2],
                   [-2, -2, 0, 2, 2],
                   [-2, 0, 2, 2, 2],
                   [0, 2, 2, 2, 2]])
emboss = cv2.filter2D(image, -1, kernel)
emboss = cv2.cvtColor(emboss, cv2.COLOR_BGR2GRAY)
cv2.imwrite('emboss.jpg', emboss)

```

由此可见，卷积滤波是图像处理中非常重要的工具，使用简单的几行代码就能实现许多 Photoshop 能够达到的效果。

4.2.3 CNNs 中的卷积层

从图像识别的角度而言,既然卷积操作可以做到增强边缘、消除噪声等效果,那自然是可以作为图片特征提取的重要工具,比如著名的 Sobel 算子就用于边缘检测,配合 SVM 算法可以在行人检测问题上获得不错的结果。但是像 Sobel 算子这种经过利用人类经验精心设计卷积核的方式有比较大的局限性:一是应用范围有限,比如 Sobel 算子只能做边缘提取,二是人类经验有限,找到合适的卷积核的代价比较大。那既然类似 Sobel 这种算子本质上是函数,是不是能用机器学习的方法来逼近呢?答案是肯定的。卷积操作与反向传播算法相结合,就诞生了卷积神经网络。从此不再需要人造卷积核,通过大量图片让程序自己训练学习卷积核参数就可以了。

从网络结构来说,卷积层节点与全连接层节点有三点主要的不同,一是局部感知域,二是权值共享,三是多核卷积。

局部感知域是指,对于每一个计算单元来说,只需要考虑其像素位置附近的输入,并不需要与上一层的所有节点相连。这一点符合人们对于图像的理解,也就是图像是相对连续的,局部信息的组合才能构成各种线条形状。这种稀疏连接的方式可以大大减少参数的个数。对于一张 1000×1000 像素的图片来说,全连接模型中每个隐层节点都有 10 万个输入,也就有对应的 10 万个权值参数,如图 4-4 所示。而若使用 10×10 的卷积核进行卷积操作的话,每个节点只有 100 个输入,对应 100 个权值,数量级大大减少。

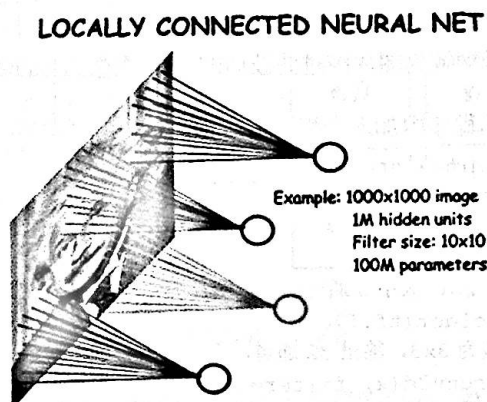


图 4-4 局部感知域

另一项减少参数个数的方式是权值共享。通过介绍卷积滤波器之后我们知道，在对一张图片进行卷积的时候，会让卷积核逐一滑过图片的每个像素，也就是说，处理每一个像素点的参数都相同。以 10×10 的卷积核卷积 1000×1000 像素的图片，最终只需要 100 个参数即可，如图 4-5 所示。

每个卷积核是一个特征提取器，若只有一个卷积核的话，就只能提取一种特征，这显然是不够的。所以使用多个卷积核，比如 32 个，同时提取 32 种特征，以多核卷积的方式保证充分提取特征。

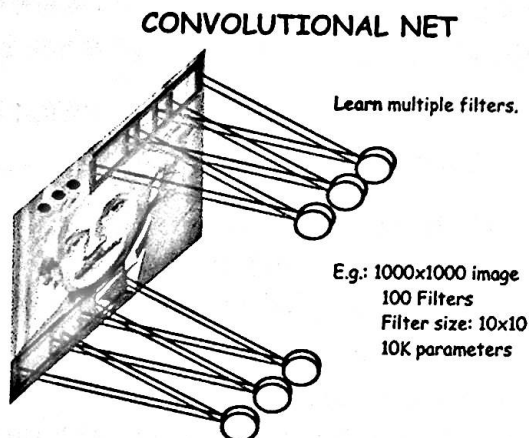


图 4-5 多核卷积

每个卷积核都会生成一幅新的图像，在卷积神经网络中，生成的图像叫做特征图 (feature maps)，可以理解为图片经过滤波后的不同通道 (channels)。

在 TensorFlow 的程序中加入卷积层是非常容易的。最常用的方法是 `tf.nn.conv2d` 方法，用于在计算图中加入 2D 卷积算子。简单的用法如下：

```
import tensorflow as tf

# 输入为 256 x 256 大小 3 通道的彩色图片
x = tf.placeholder(tf.float32, shape=[None, 256, 256, 3])
# 卷积层，卷积核为 3x3，输出 32 通道，滑动步长为 2，边缘向外填充
conv = tf.nn.conv2d(x, filter=[3, 3, 3, 32],
                    strides=[1, 2, 2, 1],
                    padding='SAME')
```


`tf.nn.conv2d` 的输入必须是 4 维 tensor，第一维是 batch，后面是图片的高 (`in_height`)、宽 (`in_width`) 和通道数 (`in_channels`)。参数 `filter` 是指定卷积核，在上面程序中指定了卷积核大小为 3×3 ，输入图像是 3 通道，输出 32 通道，即有 32 个卷积核。`strides` 指定了卷积核的滑动步长，`padding` 指定了边缘处理方式。

4.2.4 池化 (Pooling)

在经过了卷积层提取特征以后，得到的特征图代表了比像素更高级的特征，已经可以交给分类器进行训练分类了。

但是等等，好像哪里不对？现在特征数是多少？每一组卷积核都生成一幅与原图像素相同大小的特征图，节点数一个也没有减少。不仅如此，为了提取多种特征，多卷积核还会使得通道数比之前更多！这哪里降维了，分明是升维嘛！

别慌，降维的关键就在于池化操作。

池化是将图像按窗口大小划分成不重叠的区域，然后对每一个区域内的元素进行聚合。一般采用 2×2 的窗口大小，聚合方法有两种，一种是取最大值，则称为最大池化 (`max pooling`)，如图 4-6 所示，另一种是取平均值，称为平均池化 (`average pooling`)。对于窗口为 2×2 的池化操作，处理完的图像长和宽都是原图的一半，也就是说输出图的尺寸是原图的 $1/4$ 。

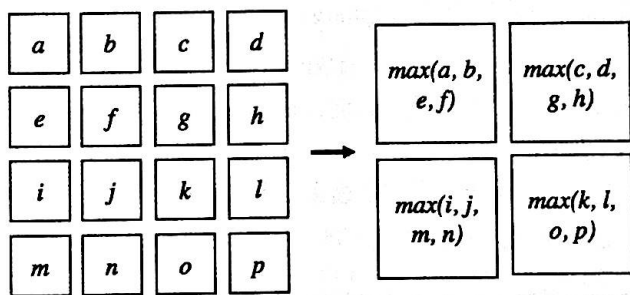


图 4-6 最大池化示意图

其实如果只考虑降维的话，不采样也能达到同样的目的。对于池化来说，更重要的是利用最大值或平均值的方法，使特征提取拥有“平移不变性” (`translation invariant`)。

也就是说,即使在图像有了几个像素的位移的情况下,依然可以获得稳定的特征集合。平移不变性对于图像识别来说具有非常重要的意义,我们更关心有哪些特征能代表目标物体本身,而不需要精确位置。举例来说,要判断两张人脸照片是否为同一个人,可能由于角度问题,五官的位置并不能精确的一一对应,即使有一定的偏移,我们还是会认为是同一个人。

卷积和池化组合在一起,为卷积神经网络加入了很强的先验经验,就是强调图片局部的连续性和相关性,同时保持平移不变性。对于图像识别来说,这样的先验经验极为有效。

TensorFlow 中原生提供了最大池化和平均池化的算子。 2×2 窗口的最大池化代码如下:

```
pool = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1],
                      strides=[1, 2, 2, 1], padding='SAME')
```

4.2.5 ReLU

在前文对卷积的介绍中提到了卷积操作实际上是一种线性操作,在计算上仅包含乘法和加法。而在机器学习领域最重要的理论基础之一,就是必须要将一个特征空间的向量通过非线性变换映射到另一个空间中才能实现线性可分。激活函数(activation function)就是引入非线性手段。

传统的 sigmoid 函数在全连接神经网络中是最常用的,是神经网络中最为核心的步骤之一。从数学角度来说, sigmoid 函数的优势是其导数的计算非常简单,使得梯度下降算法可以以极低的代价实施。然而, sigmoid 函数的劣势也同样明显,那就是只有当自变量 x 取值在 0 附近时,函数斜率才比较大,根据梯度调整参数才会有比较好的效果。而在远离中心的两侧,函数斜率快速减小并趋向于 0,导致“梯度消失”问题出现,收敛速度极慢甚至不收敛。

为了解决激活函数带来的梯度计算问题,一种更有效也更快速的激活函数被引入到神经网络中来,那就是 ReLU (Rectified Linear Units)。其函数形式非常简单:

$$f(x) = \max(0, x)$$

通过图 4-9 中展示的多种激活函数的对比可以清晰地看到, ReLU 和 sigmoid 主

要的区别在于 ReLU 的取值范围是 $[0, \infty]$ ，而 sigmoid 是 $[0, 1]$ 。因此 sigmoid 可以用于拟合概率，而 ReLU 可以将取值映射到所有正数域。

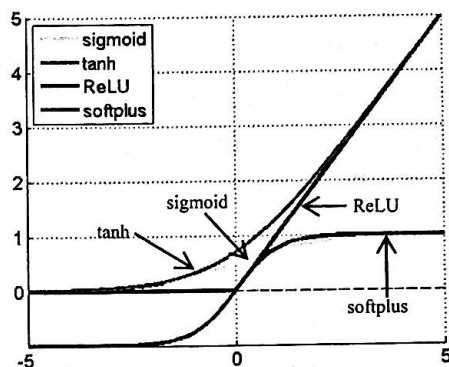


图 4-9 激活函数图形对比

从图形就可以看到，ReLU 和 sigmoid 主要的区别在于：sigmoid 是 $[0, 1]$ ，而 ReLU 的取值范围是 $[0, \infty]$ ，有更大的映射空间。

ReLU 函数的导数也极其简单，几乎不需要计算：

$$f'(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

当 x 取值为负数，就相当于直接封闭了节点。而当 x 大于 0 时，由于函数导数始终是 1，就完全避免了梯度消失的问题，保证参数能够持续收敛。AlexNet 的论文中也提到，对于同一个网络结构，使用 ReLU 作为激活函数，其收敛速度要比使用 tanh 快 6 倍以上。

在 TensorFlow 中，`tf.nn.relu` 是 ReLU 算子的实现。它只有一个参数，就是输入 tensor，tensor 的元素类型必须是数值型。此外，TensorFlow 还包含了 ReLU 的部分变种，如 `tf.nn.elu` 是实现了指数线性单元 ELU (Exponential Linear Units)，`tf.nn.relu6` 是将 ReLU 的输出限制在 $[0, 6]$ 的区间内。

分类效果好、收敛速度快、计算速度快，这些优势使得 ReLU 成为现在所有 CNN 模型首选必备的激活函数。

4.2.6 多层卷积

在第 1 章中曾经提到过,深度学习的精髓是利用多层特征组合,将简单的特征组合成复杂的抽象特征。深度卷积网络也是如此,整个网络是由多个特征提取阶段所构成的。每一个阶段都由三种操作组成:卷积、池化和非线性激活函数(ReLU)。这种组合方式能够逼近复杂的非线性模型函数,同时又能够以较简单的方式训练。实际上多个卷积层构成的网络所能表示的模型函数范围与单个卷积层的表示范围是相同的。叠加的线性函数依旧为线性函数。而非线性函数的叠加则能够极大地扩展神经网络所能表示的函数范围。

由卷积操作的定义可知。在不填充输入数据的情况下,卷积操作后图像输出结果尺寸将变小。由此可以组成一个金字塔状的结构,随着层次的加深,特征图的尺寸越来越小,但通道数变得越来越多,如图 4-7 所示。

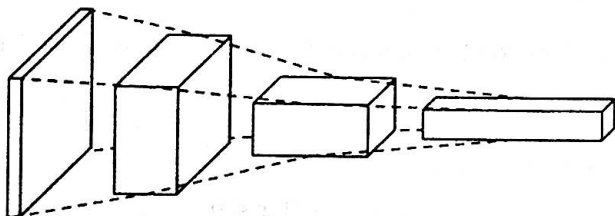


图 4-7 卷积神经网络的金字塔结构

在比较浅的层次(离原图距离近的层次),卷积会提取比较细节的特征,比如人脸图像的眼角线条、嘴唇边缘线条等。而在越深的层次,会将前面细节特征进行二次提取和组合,相当于观察的视野放大,就能提取更完整、更加抽象的特征,比如整个眼睛、整个鼻子等。最终,由全连接分类器依照最抽象的特征进行分类,就能够得到比直接处理原始像素图要好得多的结果。

4.2.7 Dropout

过拟合是指训练结果在训练集与测试集性能表现差距非常大的情况。它是一些机器学习算法的通病。这往往是因为反向传播的梯度方向与训练样本高度相关导致的。试想花费数天甚至更多的时间对一个数百 GB 的数据集进行训练,结果测试集表现很

差。这时候发现训练结果过拟合了，这时的感觉一定非常糟糕。为了削减和抵消过拟合所造成的误差，通常会采用集成学习（ensemble）的方式，将多个表现优秀的模型组合在一起进行预测，一般都会增加预测的准确度。ILSVRC 竞赛中的获胜队伍基本就都是采用这种方式。然而，对于大型深度神经网络来说，集成方式往往代价高昂，每一个模型的训练都要消耗大量的计算资源，集成后的调校更是一件具有挑战的事情。

Dropout 是深度学习领域的泰斗级科学家多伦多大学的 Hinton 教授提出的去过拟合技术。以极小的额外代价也能达到集成学习效果的方法。具体来说，就是在每一轮训练的过程中，随机让一部分隐层节点失效，这样就达到了改变网络结构的目的，但每个节点的权值都会被保留下来。在最终预测时，打开全部隐层节点，使用完整的网络进行计算，就相当于把多个不同结构的网络组合在了一起，如图 4-8 所示。

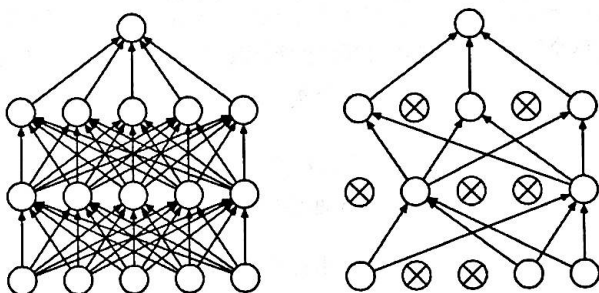


图 4-8 Dropout 在每一轮训练时都随机让部分节点失效

由于隐层节点是随机失效的，所以有 n 个隐层节点的网络理论上就会产生 2^n 个不同的网络结构。并且由于不能保证每 2 个隐含节点每次都同时出现，就削弱了节点间的联合适应性，使得权值的更新不再依赖于有固定关系的隐含节点的共同作用，增强了泛化能力。

著名的 AlexNet 将 dropout 应用在网络中的前两个全连接层。虽然需要两倍的迭代次数才能收敛，但是若不用 dropout 的话就会导致严重的过拟合。

4.3 经典 CNN 模型

自 2012 年 AlexNet 问世至今，几乎在每年的 ILSVRC 大赛上都会出现一个更加

优秀、更具有统治地位的模型出现,各种识别准确率也被这些模型一再刷新。其中最知名的模型有 AlexNet、GoogLeNet、VGGNet 和 ResNet。在这一节中,着重介绍一下这几个模型的 TensorFlow 实现。

4.3.1 AlexNet

AlexNet 由深度学习领军人物之一 Geoffrey Hinton 教授的学生 Alex Krizhevsky 提出并实现,因此以 Alex 的名字命名。AlexNet 可以说是一个具有突破性意义的模型,在它出现之前,神经网络和深度学习都陷入了长时间的瓶颈期,也越来越少被人提起,而 AlexNet 一面世,就以压倒性的成绩战胜所有对手,立刻令深度学习以气吞山河之势统治了整个图像识别领域。直至今日, AlexNet 依然是效果出色并具有很大启发意义的网络结构,我们就从这里开始,一窥 CNN 的发展。

AlexNet 的整个网络结构由 8 层神经元组成,其中前 5 层为卷积层,用于提取图形特征,后 3 层为全连接层,用于图像分类。整个网络结构一眼看上去就是我们前面所说的金字塔状结构,具体来说:

1. 输入图片是 224×224 像素的 3 通道照片;
2. 第一层使用 11×11 的卷积核,滑动步长为 4 个像素,输出 96 个特征图并进行最大池化;
3. 第二层使用 5×5 卷积核,卷积产生 256 个特征图,并进行最大池化;
4. 第三层、第四层均使用 3×3 卷积核,输出 384 个特征图;
5. 第五层使用 3×3 卷积核,输出 256 个特征图,并进行池化;
6. 第六层、第七层为全连接层,分别包含 4096 个隐层,也就是说,到全连接层时只剩 4096 个特征值;
7. 最终,第八层为 softmax 层,得到最终分类结果。

AlexNet 网络结构如图 4-10 所示。

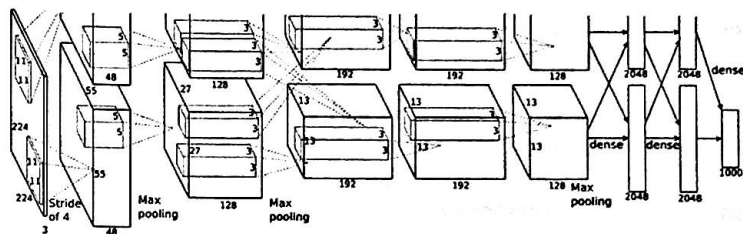


图 4-10 AlexNet 网络结构

在 AlexNet 中着重提到, ReLU 和 dropout 也起到了非常重要的作用。使用 ReLU 可以大大加快收敛速度, 比 tanh 快 6 倍。而 dropout 达到了防止模型过拟合的效果, 增强了模型的泛化能力。

AlexNet 在 ILSVRC 2012 竞赛中的 top-5 识别错误率是 15.3%, 比上一年冠军识别错误率的成绩提高了十几个百分点, 同时远超同年的第二名。

TensorFlow 官方源码中以 AlexNet 作为测试系统性能的检验程序, 并给出了 AlexNet 的代码实现¹⁷, 下面代码是其中前 5 层卷积层的模型声明代码。

```
def inference(images):
    """Build the AlexNet model.

    Args:
        images: Images Tensor

    Returns:
        pool5: the last Tensor in the convolutional component of AlexNet.
        parameters: a list of Tensors corresponding to the weights and
        biases of the
            AlexNet model.
    """
    parameters = []
    # conv1, 第一层卷积层
    with tf.name_scope('conv1') as scope:
        kernel = tf.Variable(tf.truncated_normal([11, 11, 3, 64],
            dtype=tf.float32,
            stddev=1e-1),
            name='weights')
```

¹⁷ 代码网址: <https://github.com/tensorflow/models/tree/master/tutorials/image/alexnet>.

```

conv = tf.nn.conv2d(images, kernel, [1, 4, 4, 1], padding='SAME')
biases = tf.Variable(tf.constant(0.0, shape=[64], dtype=tf.
float32),
                    trainable=True, name='biases')
bias = tf.nn.bias_add(conv, biases)
conv1 = tf.nn.relu(bias, name=scope)
print_activations(conv1)
parameters += [kernel, biases]

# lrn1
# TODO(shlens, jiajq): Add a GPU version of local response
normalization.
# 模型中应加入 local response normalization 标准化算法, 但注释中标明该
操作目前还没有 GPU 版本。

# pool1, 第一层卷积后的池化操作
pool1 = tf.nn.max_pool(conv1,
                        ksize=[1, 3, 3, 1],
                        strides=[1, 2, 2, 1],
                        padding='VALID',
                        name='pool1')
print_activations(pool1)

# conv2, 第二层卷积层
with tf.name_scope('conv2') as scope:
    kernel = tf.Variable(tf.truncated_normal([5, 5, 64, 192],
dtype=tf.float32,
                                stddev=1e-1), name='weights')
    conv = tf.nn.conv2d(pool1, kernel, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[192],
dtype=tf.float32),
                        trainable=True, name='biases')
    bias = tf.nn.bias_add(conv, biases)
    conv2 = tf.nn.relu(bias, name=scope)
    parameters += [kernel, biases]
print_activations(conv2)

# pool2, 第二层卷积后的池化操作
pool2 = tf.nn.max_pool(conv2,
                        ksize=[1, 3, 3, 1],
                        strides=[1, 2, 2, 1],
                        padding='VALID',
                        name='pool2')

```



```

print_activations(pool2)

# conv3, 第三层卷积层
with tf.name_scope('conv3') as scope:
    kernel = tf.Variable(tf.truncated_normal([3, 3, 192, 384],
                                              dtype=tf.float32,
                                              stddev=1e-1),
                          name='weights')
    conv = tf.nn.conv2d(pool2, kernel, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[384],
                                          dtype=tf.float32),
                        trainable=True, name='biases')
    bias = tf.nn.bias_add(conv, biases)
    conv3 = tf.nn.relu(bias, name=scope)
    parameters += [kernel, biases]
    print_activations(conv3)

# conv4, 第四层卷积层
with tf.name_scope('conv4') as scope:
    kernel = tf.Variable(tf.truncated_normal([3, 3, 384, 256],
                                              dtype=tf.float32,
                                              stddev=1e-1),
                          name='weights')
    conv = tf.nn.conv2d(conv3, kernel, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[256],
                                          dtype=tf.float32),
                        trainable=True, name='biases')
    bias = tf.nn.bias_add(conv, biases)
    conv4 = tf.nn.relu(bias, name=scope)
    parameters += [kernel, biases]
    print_activations(conv4)

# conv5, 第五层卷积层
with tf.name_scope('conv5') as scope:
    kernel = tf.Variable(tf.truncated_normal([3, 3, 256, 256],
                                              dtype=tf.float32,
                                              stddev=1e-1),
                          name='weights')
    conv = tf.nn.conv2d(conv4, kernel, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[256],
                                          dtype=tf.float32),
                        trainable=True, name='biases')
    bias = tf.nn.bias_add(conv, biases)

```

```
conv5 = tf.nn.relu(bias, name=scope)
parameters += [kernel, biases]
print_activations(conv5)
```

```
# pool5, 第五层卷积后的池化操作
```

```
pool5 = tf.nn.max_pool(conv5,
                        ksize=[1, 3, 3, 1],
                        strides=[1, 2, 2, 1],
                        padding='VALID',
                        name='pool5')
```

```
print_activations(pool5)
```

```
# 返回最后一层的 tensor, 和全部参数变量
```

```
return pool5, parameters
```

从这个实现看到程序写起来是比较啰嗦的, 有非常多相似的代码段。从工程的角度来说, 消除重复代码、用更少更简洁的代码实现逻辑是一贯的追求。以下就是使用 TFLearn 实现的版本, 全部代码只需要不到 40 行, 包含了对 Oxford flowers17 数据集的训练和分类, 充分体现了高级接口的易用性。

```
import tflearn
from tflearn.layers.core import input_data, dropout,
fully_connected
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.normalization import
local_response_normalization
from tflearn.layers.estimator import regression

import tflearn.datasets.oxflower17 as oxflower17
X, Y = oxflower17.load_data(one_hot=True, resize_pics=(227, 227))

# Building 'AlexNet'
# 定义输入
network = input_data(shape=[None, 227, 227, 3])
# 第一层卷积 & 池化
network = conv_2d(network, 96, 11, strides=4, activation='relu')
network = max_pool_2d(network, 3, strides=2)
network = local_response_normalization(network)
# 第二层卷积 & 池化
network = conv_2d(network, 256, 5, activation='relu')
network = max_pool_2d(network, 3, strides=2)
network = local_response_normalization(network)
```

```

# 第三、第四层卷积
network = conv_2d(network, 384, 3, activation='relu')
network = conv_2d(network, 384, 3, activation='relu')
# 第五层卷积 & 池化
network = conv_2d(network, 256, 3, activation='relu')
network = max_pool_2d(network, 3, strides=2)
network = local_response_normalization(network)
# 第一层全连接层, 以 0.5 的概率进行 dropout, 也就是每次训练只保留一半节点
network = fully_connected(network, 4096, activation='tanh')
network = dropout(network, 0.5)
# 第二层全连接层
network = fully_connected(network, 4096, activation='tanh')
network = dropout(network, 0.5)
# 第三层全连接层, softmax 输出分类结果
network = fully_connected(network, 17, activation='softmax')
# 定义优化算法、损失函数等
network = regression(network, optimizer='momentum',
                      loss='categorical_crossentropy',
                      learning_rate=0.001)

# Training
model = tflearn.DNN(network, checkpoint_path='model_alexnet',
                    max_checkpoints=1, tensorboard_verbose=2)
# 全部数据的 10% 作为验证集, 训练 1000 轮, 每一批放入 64 条样本数据
model.fit(X, Y, n_epoch=1000, validation_set=0.1, shuffle=True,
        show_metric=True, batch_size=64, snapshot_step=200,
        snapshot_epoch=False, run_id='alexnet_oxflowers17')

```

使用 TensorBoard 可以一目了然地观察完整的网络结构, 如图 4-11 所示。

AlexNet 开创了深度学习的时代, 大数据、GPU、ReLU 和 dropout 等技术都为它的出现奠定了坚实的基础。虽然现在实际应用中基本已经见不到 AlexNet, 但我们依然能从它的设计中吸收很多重要的思想, 比如要在增加网络复杂度的同时加入防止过拟合的措施。

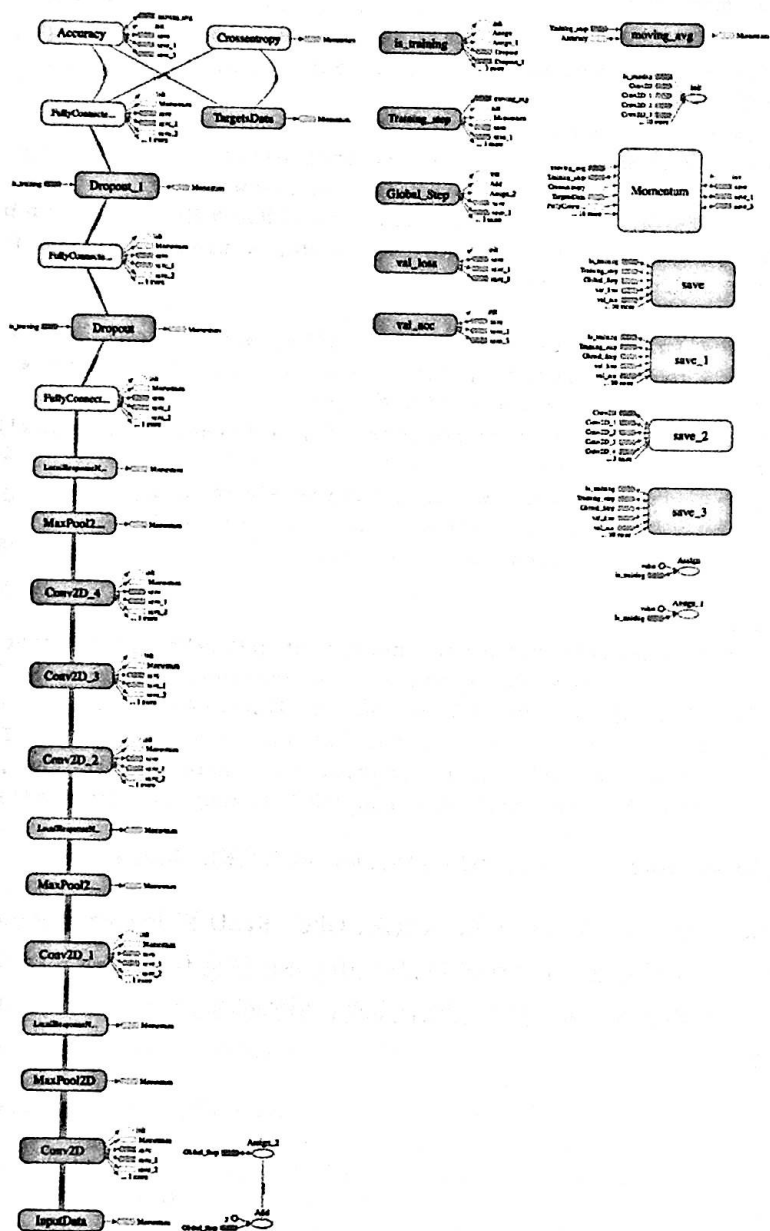


图 4-11 TensorBoard 根据代码绘制的 AlexNet 结构

4.3.2 VGGNets

VGGNets 网络结构诞生自牛津大学视觉几何组 (Visual Geometry Group), 也因此依照传统以作者 VGG 命名。VGGNet 在 ILSVRC 2014 上取得了图像分类第二名、图像定位第一名的成绩。

从网络设计思路上来说, VGGNet 是继承了 AlexNet 的思路, 以 AlexNet 为基础, 尝试建立了一个层次更多、深度更深的网络。其网络结构一样可以由 8 个层次所构成, 也是 5 组卷积层、3 层全连接层。最主要的区别在于, VGGNet 的每个卷积层并不是只做一次卷积操作, 而是连续卷积 2~4 次。结构上的差别可以参见表 4-1。

表 4-1 AlexNet 与 VGGNets 的网络结构对比

AlexNet (8 个参数层)	VGG16 (16 个参数层)	VGG19 (19 个参数层)
输入 224×224 像素 RGB 图片		
conv11-96	conv3-64 conv3-64	conv3-64 conv3-64
max pooling		
conv5-256	conv3-128 conv3-128	conv3-128 conv3-128
max pooling		
conv3-384	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
	max pooling	
conv3-384	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
	max pooling	
conv3-256	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
max pooling		
full connected-4096		
full connected-4096		
full connected-1000		
softmax		

VGG16 和 VGG19 是其论文介绍效果最好的网络结构。VGG19 的意思是模型中有 19 层计算含参数并可以被训练，包括 16 层卷积层和 3 层全连接层。

TFLearn 版本实现的 VGG19 也可以分类 Oxford flowers 17，代码如下：

```
import tflearn
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.estimator import regression

# Data loading and preprocessing
import tflearn.datasets.oxflower17 as oxflower17
X, Y = oxflower17.load_data(one_hot=True)

# Building 'VGG Network'
network = input_data(shape=[None, 224, 224, 3])

network = conv_2d(network, 64, 3, activation='relu')
network = conv_2d(network, 64, 3, activation='relu')
network = max_pool_2d(network, 2, strides=2)

network = conv_2d(network, 128, 3, activation='relu')
network = conv_2d(network, 128, 3, activation='relu')
network = max_pool_2d(network, 2, strides=2)

network = conv_2d(network, 256, 3, activation='relu')
network = conv_2d(network, 256, 3, activation='relu')
network = conv_2d(network, 256, 3, activation='relu')
network = max_pool_2d(network, 2, strides=2)

network = conv_2d(network, 512, 3, activation='relu')
network = conv_2d(network, 512, 3, activation='relu')
network = conv_2d(network, 512, 3, activation='relu')
network = max_pool_2d(network, 2, strides=2)

network = conv_2d(network, 512, 3, activation='relu')
network = conv_2d(network, 512, 3, activation='relu')
network = conv_2d(network, 512, 3, activation='relu')
network = max_pool_2d(network, 2, strides=2)

network = fully_connected(network, 4096, activation='relu')
network = dropout(network, 0.5)
```

```

network = fully_connected(network, 4096, activation='relu')
network = dropout(network, 0.5)
network = fully_connected(network, 17, activation='softmax')

network = regression(network, optimizer='rmsprop',
                      loss='categorical_crossentropy',
                      learning_rate=0.001)

# Training
model = tflearn.DNN(network, checkpoint_path='model_vgg',
                    max_checkpoints=1, tensorboard_verbose=0)
model.fit(X, Y, n_epoch=500, shuffle=True,
        show_metric=True, batch_size=32, snapshot_step=500,
        snapshot_epoch=False, run_id='vgg_oxflowers17')

```

VGG 官方公布了 caffe 和 matlab 版本的定义和参数，可以直接下载¹⁸。使用训练好的参数，一方面可以直接用于物体检测应用，另一方面也可以构造自己的模型，使用迁移学习（transfer learning）的技术，将成果应用于其他的图像问题。使用参数构造的图像特征提取网络的实现如下：

```

def vgg19(input_image):
    layers = (
        'conv1_1', 'conv1_2', 'pool1',
        'conv2_1', 'conv2_2', 'pool2',
        'conv3_1', 'conv3_2', 'conv3_3', 'conv3_4', 'pool3',
        'conv4_1', 'conv4_2', 'conv4_3', 'conv4_4', 'pool4',
        'conv5_1', 'conv5_2', 'conv5_3', 'conv5_4', 'pool5',
    )
    params = loadmat('imagenet-vgg-verydeep-19.mat')
    weights = params['layers'][0]
    network = input_image
    for i, name in enumerate(layers):
        layer_type = name[:4]
        if layer_type == 'conv':
            kernels, bias = weights[i][0][0][0][0]
            # matconvnet weights: [width, height, in_channels,
            out_channels]
            # tensorflow weights: [height, width, in_channels,
            out_channels]
            kernels = np.transpose(kernels, (1, 0, 2, 3))

```

¹⁸ VGGNets 网络参数下载网址：http://www.robots.ox.ac.uk/~vgg/research/very_deep/。

```

conv = tf.nn.conv2d(network, tf.constant(kernels),
                      strides=(1, 1, 1, 1), padding='SAME',
                      name=name)
network = tf.nn.bias_add(conv, bias.reshape(-1))
network = tf.nn.relu(network)
elif layer_type == 'pool':
    network = tf.nn.max_pool(network, ksize=(1, 2, 2, 1),
                              strides=(1, 2, 2, 1),
                              padding='SAME')

return network

```

VGG 最主要的贡献就是展示了当卷积网络的深度不断加深时，是可以将识别准确度提高到更高水平的。并且，全部使用 3×3 的卷积核也可以在保证特征提取效果的同时减少参数数量，使计算代价更小、收敛速度更快。

4.3.3 GoogLeNet & Inception

GoogLeNet 是另一个在 ILSVRC 2014 上大放异彩的模型，从命名就能看出来是出自 Google 之手。它与 VGGNet 在 ILSVRC 2014 上可谓平分秋色，是图像分类问题的冠军。

通过 VGGNets 的介绍我们了解到，如果网络的层数更多、深度更深，就会得到更好的结果。但是随着模型越来越复杂、参数越来越多，也会面临很多问题。一方面是更复杂的网络需要更多的数据才能有更好的效果，否则就比较容易过拟合。另一方面，复杂的网络意味着更大的计算量，这对于应用来说是非常不利的。在一些对实时性要求非常高的应用中，比如自动驾驶，要求参数足够少、计算速度足够快。所以，减少参数也是一个重要的课题。因此，为了能更有效地扩展网络的复杂度，Google 的大神们启动了 Inception 项目，GoogLeNet 就是 Inception 的第 1 个版本。目前 Inception 已经发布了 4 个版本，下面进行逐一介绍。

正如前面在对比 AlexNet 和 VGGNets 的结构时提到的，对于卷积核大小的选择是需要经验和大量实验才能确定的，到底是选 3×3 呢，还是 5×5 或者 7×7 ？并没有一种明确的思路。Inception 的做法是跳出直线加深网络层数的思路，通过增加“宽度”的方式增加网络的复杂度，避免陷入卷积核选择的陷阱，让程序自己学习如何选择卷积核。具体来说，是在每一个卷积层中，并行使用 1×1 卷积、 3×3 卷积、 5×5 卷积和池化，同时提取不同尺度的特征，然后通过 1×1 的卷积对每一个分支进行降维后，

最后将结果合并拼接在一起。直观地看起来，好像结构复杂了很多，本来只要一两个卷积就能完成的计算，现在却要使用四五种不同的操作。但是仔细分析可以发现，这样的设计不仅减少了参数的数量，而且由于增加了网络的“宽度”，网络对多种尺度特征的适应性更好了。GoogLeNet 中的 block 结构如图 4-12 所示。

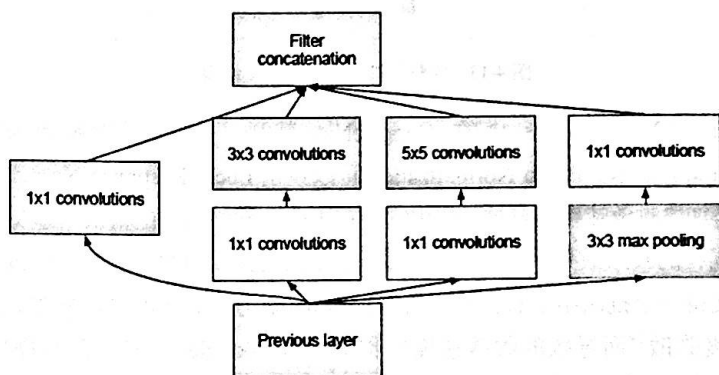


图 4-12 GoogLeNet 中的 block 结构

在这个结构中， 1×1 卷积扮演了非常重要的角色。 1×1 卷积并没有对图像本身产生什么影响，在数学上仅仅是最简单的矩阵乘法操作，其最重要的作用在于降低特征图的数量以达到降维的目的。由于有了 1×1 卷积的存在，才使得网络在不增加参数数量级的情况下可以增加复杂度。

GoogLeNet 当年在图像分类问题上的准确率无出其右，top-5 错误率只有 6.67%。相比于 VGGNets，GoogLeNet 在内存和计算消耗方面都有非常大的优势。AlexNet 有超过 6000 万个权值参数，VGGNets 的参数则是 AlexNet 的三倍以上，而 GoogLeNet 只有 500 万个参数。所以在内存受限的环境中，比如移动设备上，GoogLeNet 具有更广阔的应用空间。完整的 GoogLeNet 模型结构如图 4-13 所示。

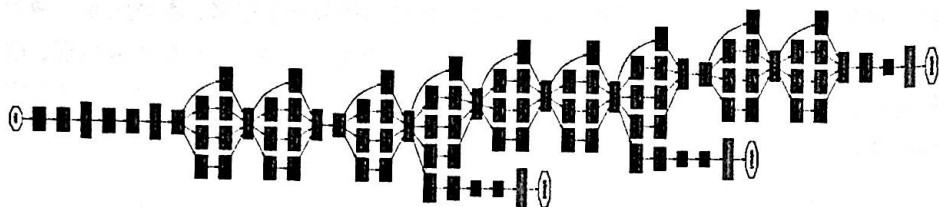


图 4-13 完整的 GoogLeNet 模型结构

Inception-v2 是在第一代的 GoogLeNet 基础上加入了批标准化 (Batch Normalization) 技术。Batch Normalization 可以说是 Local Response Normalization 的升级版,是加快收敛速度的利器,使用它能在不改变网络结构的情况下以更少的迭代次数达到同样的收敛效果。其具体做法是,对 mini-batch 中的所有信号量进行统一的归一化,使得一个批次中所有信号量符合均值为 0、方差为 1 的高斯分布。这样可以减少因梯度弥散¹⁹而导致的收敛速度缓慢。作为 Google 自家原创的操作, batch normalization 自然也是 TensorFlow 的官方标配。在代码中直接使用 `tf.nn.batch_normalization` 就可以加入该算子。一般在使用时, batch-norm 要在激活函数之前,否则其作用会打一定的折扣。

Inception 系列的第 3 版 inception-v3 在之前的版本上又有提高。其最核心的思想是将卷积操作继续分解成更小的卷积核。首先,借鉴 VGGNets 的思路, 5×5 的卷积可以由连续 2 层 3×3 卷积所替代,这样既减少了参数数量,也进一步加快了计算速度。其次, 3×3 的卷积也还可以再分解为非对称的 1×3 和 3×1 两层卷积操作。在经过这样的转换之后,不但参数数量再次减少,计算速度更快,而且网络的深度也加深了,增加了非线性表达能力。同样以 ImageNet 数据集作为检验标准, inception-v3 的 top-1 错误率只有 17.2%, top-5 错误率更是只有惊人的 3.58%,真正是比人类还要精准!

TensorFlow 的官方教程中演示了 inception-v3 模型的使用方法²⁰并配有教程文档²¹。代码中只有一个文件 `classify_image.py`,该程序运行已经训练好的 inception-v3 对图片

19 梯度弥散: 当使用反向传播计算导数的时候,随着传播层数的增多,误差增大,梯度急剧减小,导致网络最初几层参数的修正幅度趋近于 0,不能从样本中得到充分学习。

20 Inception-v3 示例代码: <https://github.com/tensorflow/models/tree/master/tutorials/image/imagenet>。

21 官方教程文档: https://tensorflow.org/tutorials/image_recognition/。

进行分类，判断图像物体的类别。

```

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import os.path
import re
import sys
import tarfile

import numpy as np
from six.moves import urllib
import tensorflow as tf

FLAGS = None

# pylint: disable=line-too-long
DATA_URL = 'http://download.tensorflow.org/models/image/imagenet/
inception-2015-12-05.tgz'
# pylint: enable=line-too-long

# NodeLookup 类的作用是将分类器输出的类别编号与人类可读的标签名称对应起来
# 例如分类 1 对应的类别编号为 n02119789，文字标签为"kit fox, Vulpes
macrootis"

class NodeLookup(object):
    """Converts integer node ID's to human readable labels."""

    def __init__(self,
                  label_lookup_path=None,
                  uid_lookup_path=None):
        if not label_lookup_path:
            label_lookup_path = os.path.join(
                FLAGS.model_dir,
                'imagenet_2012_challenge_label_map_proto.pbtxt')
        if not uid_lookup_path:
            uid_lookup_path = os.path.join(
                FLAGS.model_dir,
                'imagenet_synset_to_human_label_map.txt')
        self.node_lookup = self.load(label_lookup_path, uid_lookup_
path)

```

```
def load(self, label_lookup_path, uid_lookup_path):
    """Loads a human readable English name for each softmax node.

    Args:
        label_lookup_path: string UID to integer node ID.
        uid_lookup_path: string UID to human-readable string.

    Returns:
        dict from integer node ID to human-readable string.
    """
    if not tf.gfile.Exists(uid_lookup_path):
        tf.logging.fatal('File does not exist %s', uid_lookup_path)
    if not tf.gfile.Exists(label_lookup_path):
        tf.logging.fatal('File does not exist %s', label_lookup_path)

    # Loads mapping from string UID to human-readable string
    proto_as_ascii_lines = tf.gfile.GFile(uid_lookup_path).readlines()
    uid_to_human = {}
    p = re.compile(r'[n\d]*[ \S,]*')
    for line in proto_as_ascii_lines:
        parsed_items = p.findall(line)
        uid = parsed_items[0]
        human_string = parsed_items[2]
        uid_to_human[uid] = human_string

    # Loads mapping from string UID to integer node ID.
    node_id_to_uid = {}
    proto_as_ascii = tf.gfile.GFile(label_lookup_path).readlines()
    for line in proto_as_ascii:
        if line.startswith(' target_class:'):
            target_class = int(line.split(':')[1])
        if line.startswith(' target_class_string:'):
            target_class_string = line.split(':')[1]
            node_id_to_uid[target_class] = target_class_string[1:-2]

    # Loads the final mapping of integer node ID to human-readable
    string
    node_id_to_name = {}
    for key, val in node_id_to_uid.items():
        if val not in uid_to_human:
```

```

        tf.logging.fatal('Failed to locate: %s', val)
        name = uid_to_human[val]
        node_id_to_name[key] = name

    return node_id_to_name

def id_to_string(self, node_id):
    if node_id not in self.node_lookup:
        return ''
    return self.node_lookup[node_id]

# 从protocol buffer 文件中反序列化出 inception-v3 模型及参数
def create_graph():
    """Creates a graph from saved GraphDef file and returns a saver."""
    # Creates graph from saved graph_def.pb.
    with tf.gfile.FastGFile(os.path.join(
        FLAGS.model_dir, 'classify_image_graph_def.pb'), 'rb') as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())
        _ = tf.import_graph_def(graph_def, name='')

# 使用模型对 image 图片进行分类, 输出 top-5 置信度的类别预测
def run_inference_on_image(image):
    """Runs inference on an image.

    Args:
        image: Image file name.

    Returns:
        Nothing
    """
    if not tf.gfile.Exists(image):
        tf.logging.fatal('File does not exist %s', image)
    image_data = tf.gfile.FastGFile(image, 'rb').read()

    # Creates graph from saved GraphDef.
    create_graph()

    with tf.Session() as sess:
        # Some useful tensors:
        # 'softmax:0': A tensor containing the normalized prediction
        across
        # 1000 labels.

```

```

# 'pool_3:0': A tensor containing the next-to-last layer
containing 2048
# float description of the image.
# 'DecodeJpeg/contents:0': A tensor containing a string
providing JPEG
# encoding of the image.
# Runs the softmax tensor by feeding the image_data as input to
the graph.
softmax_tensor = sess.graph.get_tensor_by_name('softmax:0')
predictions = sess.run(softmax_tensor,
                        {'DecodeJpeg/contents:0': image_data})
predictions = np.squeeze(predictions)

# Creates node ID --> English string lookup.
node_lookup = NodeLookup()

top_k = predictions.argsort()[-FLAGS.num_top_predictions:][::-1]
for node_id in top_k:
    human_string = node_lookup.id_to_string(node_id)
    score = predictions[node_id]
    print('%s (score = %.5f)' % (human_string, score))

# 下载模型存档并解压
def maybe_download_and_extract():
    """Download and extract model tar file."""
    dest_directory = FLAGS.model_dir
    if not os.path.exists(dest_directory):
        os.makedirs(dest_directory)
    filename = DATA_URL.split('/')[-1]
    filepath = os.path.join(dest_directory, filename)
    if not os.path.exists(filepath):
        def _progress(count, block_size, total_size):
            sys.stdout.write('\r>> Downloading %s %.1f%%' % (
                filename, float(count * block_size) / float(total_size) *
100.0))
            sys.stdout.flush()
        filepath, _ = urllib.request.urlretrieve(DATA_URL, filepath,
        _progress)
        print()
        statinfo = os.stat(filepath)
        print('Successfully downloaded', filename, statinfo.st_size,
        'bytes.')
    tarfile.open(filepath, 'r:gz').extractall(dest_directory)

```

```
def main(_):
    maybe_download_and_extract()
    image = (FLAGS.image_file if FLAGS.image_file else
             os.path.join(FLAGS.model_dir, 'cropped_panda.jpg'))
    run_inference_on_image(image)
```

上述程序会先从网上下载 inception-v3 模型的存档文件，其中包含计算图 GraphDef 的序列化文件，以及模型分类编号与人类可读的文字标注的映射表。直接运行程序的话，会对一张萌萌的熊猫图片进行识别。

```
$ python classify_image.py
>> Downloading inception-2015-12-05.tgz 100.0%
Successfully downloaded inception-2015-12-05.tgz 88931400 bytes.

giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca
(score = 0.89233)
indri, indris, Indri indri, Indri brevicaudatus (score = 0.00859)
lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens
(score = 0.00264)
custard apple (score = 0.00141)
earthstar (score = 0.00107)
```

结果显示，图片是大熊猫的可能性是89.2%，狐猴的可能性是0.8%，小熊猫的可能性是0.2%，南美番荔枝或地星（一种真菌）的可能性为0.1%。模型预测图片是其他类别的概率比0.1%更小，没有显示。

在感受模型识别能力强劲的同时，在这个示例代码中还有一点值得注意的是，`create_graph()`函数展示了 TensorFlow 的模型可以直接通过 protocol buffer 格式文件的反序列化得到。本例中，模型被完整保存在 `classify_image_graph_def.pb` 文件中，反序列化成 GraphDef 对象，设置为系统默认计算图就可以使用了。这个过程中有两个比较重要的函数，`Graph.as_graph_def()`可以得到计算图对象序列化后的字节流，而 `GraphDef.ParseFromString()`方法可以从缓冲区中反序列化出 GraphDef 对象。这种方式使用起来是非常便利的，代码中并不需要再声明一遍网络结构，节省了代码调试时间，也避免了不必要的错误。

Inception-v4 是 Inception 系列的最新版本，在设计上充分借鉴了 ResNet 残差网

络,在大大提高训练速度的同时也进一步提升了预测准确率。接下来就介绍解决了梯度弥散问题的 ResNet。

4.3.4 ResNets

回顾 2016 年计算机视觉领域的发展成果,微软亚洲研究院的何恺明博士及他所在的微软亚洲研究院 MSRA 团队推出的深度残差网络 ResNets 是最大的亮点。其最大的贡献就是通过设计一种残差网络的结构,避免了随着网络层数加深而产生的梯度消失或梯度爆炸的问题,不但使深度神经网络的收敛速度更快、精度更高,而且让加深网络深度成为可能。

对于深度神经网络来说, VGGNets 证明了加深网络层次是提高精度的有效手段,但是由于梯度弥散的问题导致网络深度无法持续加深。梯度弥散问题是由于在反向传播过程中误差不断累积,导致在最初的几层梯度值几乎为 0,从而无法收敛。经过测试,20 层以上的深层网络,会随着层数的增加,收敛效果越来越差,50 层的网络是 20 层网络所得错误率的一倍。何恺明博士将这一现象称为深度网络的退化问题 (degradation problem)。

退化问题其实说明,不是所有的系统都能很容易地被优化。难道网络的深度的增加就到此为止了么? ResNets 告诉我们残差网络是一种避免梯度消失的更容易优化的结构。

熟悉机器学习算法的都能理解,神经网络实际上是将一个空间维度的向量 x , 经过非线性变换 $H(x)$ 映射到另外一个空间维度中。但通过前面的观察会意识到 $H(x)$ 非常难以优化,所以这里尝试转而求 $H(x)$ 的残差形式 $F(x) = H(x) - x$ 。假设求解 $F(x)$ 会比求 $H(x)$ 要简单一些的话,就可以通过 $F(x) + x$ 来达到最终的目标。图 4-14 展示了残差网络的单元结构,称之为残差块 (residual block)。

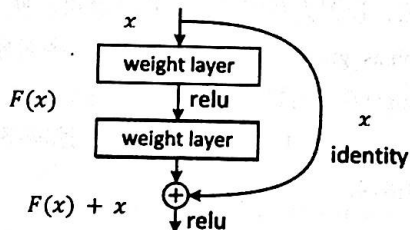


图 4-14 残差块 residual block

从实际实验的结果来看, 残差网络的效果是非常明显的。如论文里所说, 类似 VGGNet 的结构在超过 20 个参数层以后, 收敛效果会大打折扣, 准确率比较差, 但是简单地加入一些“捷径”(shortcut)连接边后, 使其转变为残差结构, 收敛效果都极具提高, 精度也随着训练次数的增加持续提高, 并且不断加深网络深度还可以继续提高准确率, 残差网络与 VGG 结构的对比如图 4-15 所示。

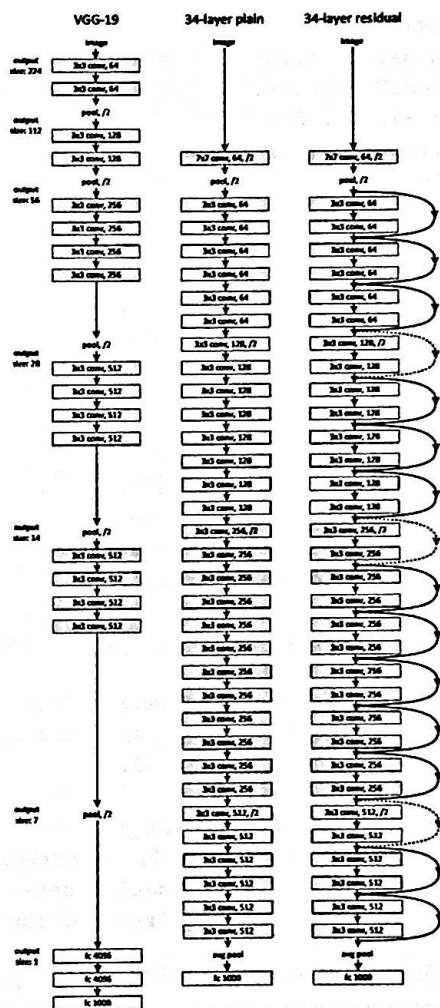


图 4-15 残差网络与 VGG 结构对比

凭借出色的表现，ResNet 在 ILSVRC 2015 竞赛中一举斩获了 5 个项目的冠军，并以图像分类 top-5 错误率仅 3.57% 的成绩傲视群雄。

TFlearn 版本的 ResNets 程序示例如下，可以用于处理 mnist 或者 cifar10 数据集。

```
import tflearn
import tflearn.data_utils as du

# 加载数据并做预处理
import tflearn.datasets.mnist as mnist
X, Y, testX, testY = mnist.load_data(one_hot=True)
X = X.reshape([-1, 28, 28, 1])
testX = testX.reshape([-1, 28, 28, 1])
X, mean = du.featurewise_zero_center(X)
testX = du.featurewise_zero_center(testX, mean)

# 构建残差网络模型
net = tflearn.input_data(shape=[None, 28, 28, 1])
net = tflearn.conv_2d(net, 64, 3, activation='relu', bias=False)
# 使用 Bottleneck 结构构建残差块
net = tflearn.residual_bottleneck(net, 3, 16, 64)
net = tflearn.residual_bottleneck(net, 1, 32, 128, downsample=True)
net = tflearn.residual_bottleneck(net, 2, 32, 128)
net = tflearn.residual_bottleneck(net, 1, 64, 256, downsample=True)
net = tflearn.residual_bottleneck(net, 2, 64, 256)
net = tflearn.batch_normalization(net)
net = tflearn.activation(net, 'relu')
net = tflearn.global_avg_pool(net)
net = tflearn.fully_connected(net, 10, activation='softmax')
# 声明优化算法、损失函数、学习率等
net = tflearn.regression(net, optimizer='momentum',
                           loss='categorical_crossentropy',
                           learning_rate=0.1)

# 训练
model = tflearn.DNN(net, checkpoint_path='model_resnet_mnist',
                     max_checkpoints=10, tensorboard_verbose=0)
model.fit(X, Y, n_epoch=100, validation_set=(testX, testY),
         show_metric=True, batch_size=256, run_id='resnet_mnist')
```

ResNets 的成功，使整个卷积神经网络的研究上了一个新的台阶，inception 也将残差结构融入其中，实现了更优秀的模型 inception-v4。在图像风格化的应用中，我们也还会见到残差网络的运用。

4.4 图像风格转换

在 2015 年的 NIPS 会议上，一篇由德国图宾根大学的 Leon A. Gatys 等人发表的名为 *A Neural Algorithm of Artistic Style* 的论文引起了大家极大的兴趣。Gatys 的工作是运用深度学习的方法将普通的照片与艺术画作进行融合，使照片看起来具有名画的画风。最典型的就是将普通的风景照片渲染成梵高著名作品《星夜》(*The Starry Night*) 的画风，成品图在照片内容的基础上，融合了深蓝与明黄混合的色调和《星夜》中极具代表性的漩涡般扭曲的纹理。不同于以往的滤镜，经过论文中深度学习方法的处理后，原图的画面产生了大规模的改变，为人们带来了各种梦幻般的视觉体验，如图 4-16 所示。

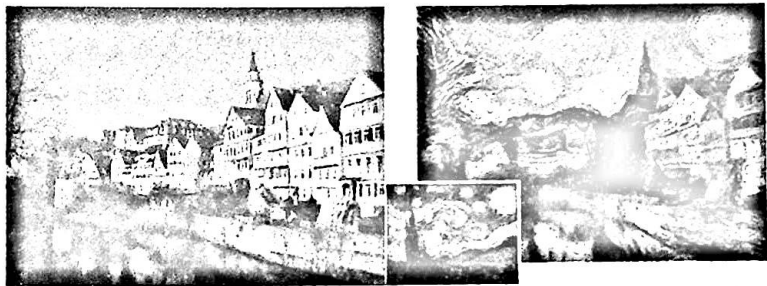


图 4-16 将风景画转换为《星夜》风格的艺术作品

其实所谓“绘画风格”，是一个十分抽象的概念。人们讨论绘画风格更多的是在描述某一个画家或者画派的作品所共同拥有的特点，不同的风格有许多不同的特征，可能是色彩，可能是笔触，也可能是图画表达的深层次的内涵。那这种无法准确描述的概念，让以数学为基础的计算机如何模仿？Gatys 在论文中给出了一种很有趣的答案——捕捉画中的纹理信息。从上面《星夜》的例子中就能看出，只要是颜色和线条符合了画的特征，就可以让人很明显地感觉到有画作的风格。

4.4.1 量化的风格

通过前面章节的介绍我们知道，CNNs 可以通过多层卷积操作来提取物体的抽象特征，并且在物体识别方面取得了惊人的成绩。而 Gatys 等作者则是另辟蹊径，开辟了另一个深度卷积网络的应用场景——纹理合成 (texture synthesis) 和风格迁移 (style transfer)。

对于一个已经训练好的深度卷积网络来说,每一个卷积操作所得的特征图都是对图像中某方面的特征的提取结果。随着层次的加深,网络提取出的特征更抽象,特征数量也更少,更能代表物体本身。因此,网络的高层所得到的特征结果集合,可以认为是图像内容的量化表示。

在 Gatys 的论文中,定义了神奇的格拉姆矩阵 (Gram matrix),用于捕获了绘画中的纹理特征。格拉姆矩阵是一组向量的内积的对称矩阵,在这里的计算方法是将所有特征图矩阵矢量化后,求其内积之和,它代表了不同滤波器所得结果之间的相关性。在这种高阶的统计方法之上,像素级的特征全部都被丢弃,甚至全局的场景信息也都没有保留,仅存了高级的绘画风格。在这种量化方法下,不同种类的纹理得以区分,甚至可以被逼近。

图 4-17 用更直观的方式表现了深度卷积网络在不同阶段所提取的特征的区别。对于 VGGNet 这样的 5 层的卷积网络,特征图的尺寸会越来越小,通道数越来越多。通过反卷积之类的图像重建操作,可以将不同层次的特征图重建成一幅完整的图像。对于层次较浅(靠近输入)的层,如图中的 a, b, c 层,其重建的图像与原图看不出有多大的差别。而对于更深的层重建的图像,就与原图有较大的差别,虽然还能看出是房屋的图案,但像素都已经被打散,不再是规则的图形。另一方面,将《星夜》放入卷积网络,通过纹理合成 (Texture synthesis) 等重建方法也生成一系列的图,则可以看到在比较浅的层次主要还是细碎的、颜色鲜明的点,随着层次的加深,逐渐出现了大片的、具有显著风格的形状和纹理。

既然图像内容物体和绘画风格都已经有了量化表示,那将二者融合就成了一个优化问题。合成图需要在内容维度上必须逼近照片,原图中是房屋,在合成图中依然要看得出是房屋。同时在风格维度上逼近绘画,要以《星夜》中最具代表性的颜色和纹理布满整个画面。Gatys 等人的这篇论文的核心,就是利用预先训练好的 VGGNet 来提取图片中内容及风格的数值化特征,然后定义了一种特殊的损失函数来评估合成图片符合“风格”的程度,再然后使用梯度下降的方法来不断修正合成图的各个像素以使损失值变小。当损失值最小的时候,便是我们期望的“名画版照片”。

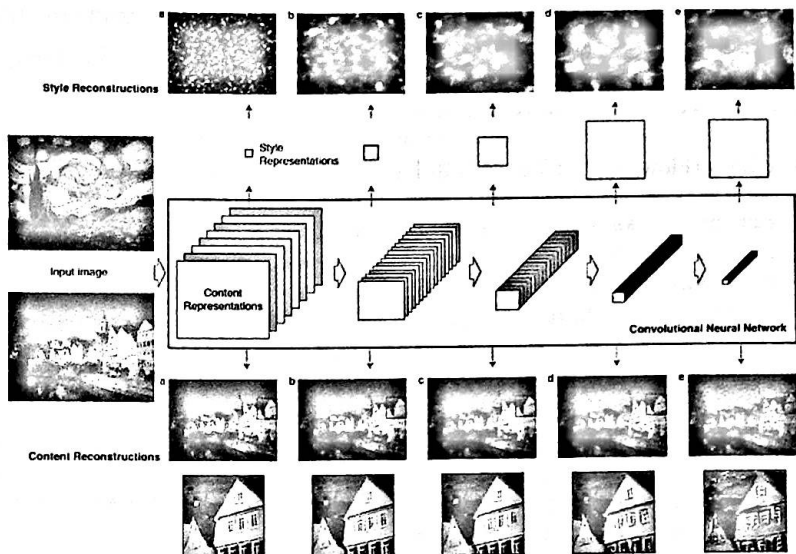


图 4-17 VGG 网络在风格和内容量化提取中的作用

具体来说，损失函数是基于 VGGNet 中的特征图来构建的。损失函数由两部分构成，一部分是内容损失 $\mathcal{L}_{\text{content}}$ ，另一部分是风格损失 $\mathcal{L}_{\text{style}}$ 。其中，内容损失 $\mathcal{L}_{\text{content}}$ 为照片与合成图在 VGG19 中 relu4_2 层输出的特征图中每个元素的差的平方和。

$$\mathcal{L}_{\text{content}}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F_{ij}^l - P_{ij}^l)^2$$

其中 \vec{p} 和 \vec{x} 分别代表原始内容图和生成图， F^l 和 P^l 表示它们在 l 层的特征图， l 实际上代表 VGG19 中的 relu4_2 层。

风格损失 $\mathcal{L}_{\text{style}}$ 是风格图与合成图以 relu1_1, relu2_1, relu3_, relu4_1, relu5_1 五层的特征图所求得的格拉姆矩阵的差的平方和。

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$$

$$\mathcal{L}_{\text{style}}(\vec{a}, \vec{x}) = \sum_{l=0}^L w_l E_l$$

其中 \tilde{a} 和 \tilde{x} 表示风格图和生成图, A^l 和 G^l 表示它们在 l 层特征图的格拉姆矩阵, N 和 M 为格拉姆矩阵的边长。 w_l 为各层风格损失设置的权重, 正如前文提到的, 浅层更注重纹理细节, 深层更注重大片的图案。

使用 TensorFlow 实现的完整版本如下:

```
import numpy as np
import scipy.io
import tensorflow as tf
from PIL import Image

# 定义命令行参数
tf.app.flags.DEFINE_string('style_image', '', 'style image')
tf.app.flags.DEFINE_string('content_image', '', 'content image')
tf.app.flags.DEFINE_integer('epochs', 5000, 'training epochs')
tf.app.flags.DEFINE_float('learning_rate', 0.5, 'learning rate')
FLAGS = tf.app.flags.FLAGS

# 声明超参数
STYLE_WEIGHT = 1.
CONTENT_WEIGHT = 0.
STYLE_LAYERS = ['relu1_1', 'relu2_1', 'relu3_1', 'relu4_1', 'relu5_1']
CONTENT_LAYERS = ['relu4_2']
_vgg_params = None

def vgg_params():
    # 加载 VGG19 的权值
    global _vgg_params
    if _vgg_params is None:
        _vgg_params = scipy.io.loadmat('imagenet-vgg-verydeep-19.mat')
    return _vgg_params

def vgg19(input_image):
    # 声明 VGG19 网络结构,
    layers = (
        'conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1',
        'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'pool2',
        'conv3_1', 'relu3_1', 'conv3_2', 'relu3_2', 'conv3_3',
        'relu3_3', 'conv3_4', 'relu3_4', 'pool3',
        'conv4_1', 'relu4_1', 'conv4_2', 'relu4_2', 'conv4_3',
```

```

        'relu4_3', 'conv4_4', 'relu4_4', 'pool4',
        'conv5_1', 'relu5_1', 'conv5_2', 'relu5_2', 'conv5_3',
        'relu5_3', 'conv5_4', 'relu5_4', 'pool5'
    )
    weights = vgg_params()['layers'][0]
    net = input_image
    network = {}
    for i, name in enumerate(layers):
        layer_type = name[:4]
        if layer_type == 'conv':
            kernels, bias = weights[i][0][0][0][0]
            # matconvnet weights: [width, height, in_channels,
out_channels]
            # tensorflow weights: [height, width, in_channels,
out_channels]
            kernels = np.transpose(kernels, (1, 0, 2, 3))
            conv = tf.nn.conv2d(net, tf.constant(kernels),
                                strides=(1, 1, 1, 1), padding='SAME',
                                name=name)
            net = tf.nn.bias_add(conv, bias.reshape(-1))
            net = tf.nn.relu(net)
        elif layer_type == 'pool':
            net = tf.nn.max_pool(net, ksize=(1, 2, 2, 1),
                                strides=(1, 2, 2, 1),
                                padding='SAME')
        network[name] = net
    return network

def content_loss(target_features, content_features):
    # 使用特征图之差的平方和作为内容差距，越小则合成图与原图的内容越相近
    _, height, width, channel = map(lambda i: i.value,
                                     content_features.get_shape())
    content_size = height * width * channel
    return tf.nn.l2_loss(target_features - content_features) /
content_size

def style_loss(target_features, style_features):
    # 使用 Gram matrix 之差的平方和作为风格差距，越小则合成图越具有风格图的
纹理特征
    _, height, width, channel = map(lambda i: i.value,
                                     target_features.get_shape())
    size = height * width * channel
    # target_gram 是特征图矩阵的内积

```

```

target_features = tf.reshape(target_features, (-1, channel))
target_gram = tf.matmul(tf.transpose(target_features),
                        target_features) / size
style_features = tf.reshape(style_features, (-1, channel))
style_gram = tf.matmul(tf.transpose(style_features),
                      style_features) / size
return tf.nn.l2_loss(target_gram - style_gram) / size

def loss_function(content_image, style_image, target_image):
    # 总损失 = 内容损失 * 内容权重 + 风格损失 * 风格权重
    style_features = vgg19([style_image])
    content_features = vgg19([content_image])
    target_features = vgg19([target_image])
    loss = 0.0
    for layer in CONTENT_LAYERS:
        loss += CONTENT_WEIGHT * content_loss(target_features[layer],
                                              content_features[layer])
    for layer in STYLE_LAYERS:
        loss += STYLE_WEIGHT * style_loss(target_features[layer],
                                           style_features[layer])
    return loss

def stylize(style_image, content_image, learning_rate=0.1,
            epochs=500):
    # 目标合成图，初始化为随机白噪声图
    target = tf.Variable(tf.random_normal(content_image.shape),
                        dtype=tf.float32)
    style_input = tf.constant(style_image, dtype=tf.float32)
    content_input = tf.constant(content_image, dtype=tf.float32)
    cost = loss_function(content_input, style_input, target)
    # 使用 Adam 算法作为优化算法，最小化代价函数
    train_op = tf.train.AdamOptimizer(learning_rate).minimize(cost)
    with tf.Session() as sess:
        tf.initialize_all_variables().run()
        for i in range(epochs):
            _, loss, target_image = sess.run([train_op, cost,
            target])

    # 打印记录迭代中损失函数下降过程
    print("iter:%d, loss:%.9f" % (i, loss))
    if (i + 1) % 100 == 0:
        # save target image every 100 iterations
        image = np.clip(target_image + 128, 0, 255).astype
(np.uint8)

```



```
Image.fromarray(image).save("neural_%d.jpg" % (i + 1))
```

```
if __name__ == '__main__':
    # 图片在读入时, 像素值被预处理为 0 中心, 可以加速收敛
    style = Image.open(FLAGS.style_image)
    style = np.array(style).astype(np.float32) - 128.0
    content = Image.open(FLAGS.content_image)
    content = np.array(content).astype(np.float32) - 128.0
    stylize(style, content, FLAGS.learning_rate, FLAGS.epochs)
```

从下面这一组图片中可以看出《星夜》的风格施加在照片上的过程。我们可以看到, 在最初的迭代中, 图片更像是原始照片和绘画的简单纹理的叠加, 而随着迭代次数的增加, 图片慢慢学习到了《星夜》中的配色和笔触的纹理, 并在随后的迭代过程中逐渐成形, 最终把一张照片转变成为梵高的风格, 如图 4-18 所示。

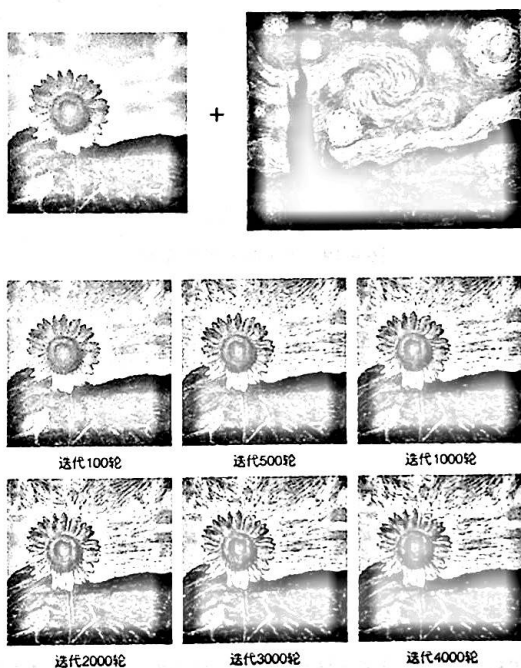


图 4-18 鲜花照片与《星夜》风格的融合过程

是不是十分有趣？还有更有趣的！

在程序中我们在最开始设置了一些超参数，如学习率、内容损失的系数、风格损失计算的特征图等，还有一些是没有在程序中定义出来的超参数，比如各个特征图系数。通过实验调整这些参数，可以对 CNN 在风格迁移这一应用上有更深的理解。比如学习率会影响收敛的速度、内容损失和风格损失的系数会影响合成图片的效果。特别来说，如果我们将内容损失的系数设置为 0，也就是只希望拟合风格图的纹理，那么《星夜》的纹理是如图 4-19 这样的。

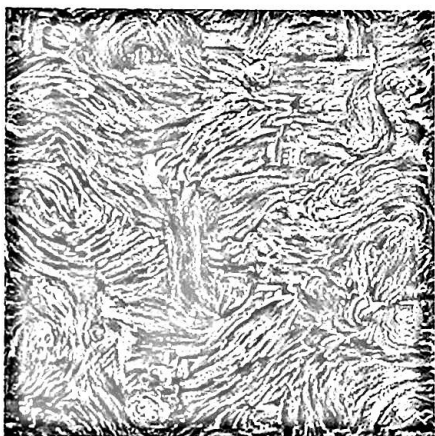


图 4-19 《星夜》风格纹理

对于普通人来说，这些线条和配色看起来与《星夜》非常相似，最大的差别是没有了绘画中的一些细节，这也正是纹理提取所要做的。

4.4.2 风格的滤镜

2016 年，一款名为 Prisma 的滤镜 APP 以病毒式传播的速度席卷了全球各大社交网络，上线 5 周时间内就俘获了 1000 万用户，成为增长速度最快的现象级产品。这款 APP 的核心功能就只有一个：多种名画风格的照片滤镜。这实际上就是我们前面所讲的风格迁移，那么这是将 neural style 论文中的方法直接实现成 APP 就放出来推广了么？并不是。Prisma 滤镜最大的特点就是不但能渲染出让人惊叹的效果，而且速度飞快！一张照片的渲染只需要几秒钟！

Gatys 论文中所描述的使用梯度下降算法逐像素修改合成图的方法, 虽然效果非常惊艳, 但是最大的问题在于每一幅图像的渲染都要经过漫长的学习过程, 在多轮迭代计算中一个像素一个像素地施加纹理。以上一小节中鲜花和《星夜》的融合为例, 经过了几千轮的迭代训练才形成了我们能识别的风格, 而每一轮迭代计算即使是在性能最强劲的 GPU Titan X 上, 依然需要数百毫秒的时间, 若运行在 CPU 上则是以分钟计。不仅如此, 在实验中的照片分辨率仅为 512×512 像素, 当照片的分辨率提升时, 计算所需要的时间还会成倍增加。也就是说要得到一张效果不错的具有名画风格的图片, 需要等待数分钟甚至数小时, 这样的处理速率是不可能应对大规模用户的使用的。Prisma 产品化最成功的一点就在于它能够在几秒钟内完成一张图片的渲染。这种速度是如何做到的?

有两篇关于风格滤镜的论文几乎在同一时间发表出来, 分别是由俄罗斯科学家 Dmitry Ulyanov 完成的 *Texture Networks: Feed-forward Synthesis of Textures and Stylized Images* 和斯坦福大学李飞飞实验室的 Justin Johnson 发表的 *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*。这两篇论文都描述了实时风格迁移 (Real-Time Style Transfer) 的方法, 其中的核心思路是: 以 neural style 的研究为基础, 基于 VGGNet 所提取出的高维抽象特征构建损失函数, 同时使用一个图像变换卷积网络来存储风格的纹理特征, 然后将训练好的网络直接作为滤镜使用即可完成对图片的风格变换。这应该就是 Prisma 产品背后的技术奥秘。

如图 4-20 所示, 整个系统由两个深度卷积网络组成, 一个图像生成网络 f_w , 以非线性变换 $y = f_w(x)$ 的方式将输入图片 x 转换为输出图片 y , 另一个是损失计算网络, 是一个训练好的确定参数的 VGGNet。图像生成网络就是我们最后需要的滤镜。

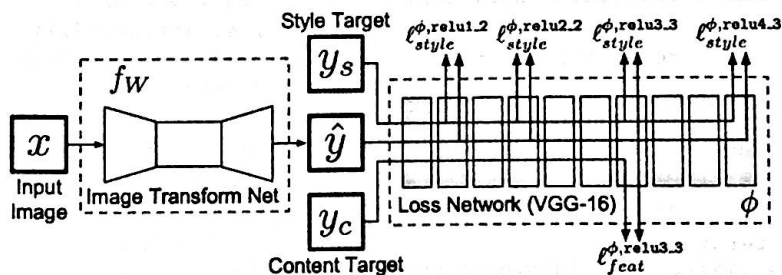


图 4-20 风格滤镜系统结构

在这样一个系统中，损失的计算与 *neural style* 完全相同，要最小化内容损失与风格损失之和。与 *neural style* 不同的是，合成图是由图像生成网络生成出来的，所以梯度下降的传播不会直接改变合成图的像素，而是修改图像生成网络的参数。针对每一种风格，都训练一个单独的图像生成网络，得到一组风格对应的参数。

Johnson 在论文中提出的是一种沙漏型结构的生成器。在前部是三层卷积，中部为 5 个残差块，后部为三层反卷积。整个网络含有约 100 万个参数，占用 4MB 内存。具体结构可以参看代码：

```
def johnson(input_image):
    relu = tf.nn.relu
    conv2d = tflearn.conv_2d

    def batch_norm(x):
        mean, var = tf.nn.moments(x, axes=[1, 2, 3])
        return tf.nn.batch_normalization(x, mean, var, 0, 1, 1e-5)

    def deconv2d(x, n_filter, ksize, strides=1):
        _, h, w, _ = x.get_shape().as_list()
        output_shape = [strides * h, strides * w]
        return tflearn.conv_2d_transpose(x, n_filter, ksize,
                                          output_shape,
                                          strides)

    def res_block(x):
        net = relu(batch_norm(conv2d(x, 128, 3)))
        net = batch_norm(conv2d(net, 128, 3))
        return x + net

    net = relu(batch_norm(conv2d(input_image, 32, 9)))
    net = relu(batch_norm(conv2d(net, 64, 4, strides=2)))
    net = relu(batch_norm(conv2d(net, 128, 4, strides=2)))
    for i in range(5):
        net = res_block(net)
    net = relu(batch_norm(deconv2d(net, 64, 4, strides=2)))
    net = relu(batch_norm(deconv2d(net, 32, 4, strides=2)))
    net = deconv2d(net, 3, 9)
    return net
```

Texture Nets 是 Ulyanov 提出的另一种图像生成网络结构，是由不同分辨率卷积

堆叠后组成的金字塔形状网络，如图 4-21 所示。这样一个轻型网络的参数个数为 65KB 左右，可以压缩到占用 300KB 内存。在 GPU 上 20 毫秒就可以处理一张 256×256 大小的图片，并且仅需要 170MB 的内存空间，相比而言原始 neural style 的方法需要用超过 1100MB。

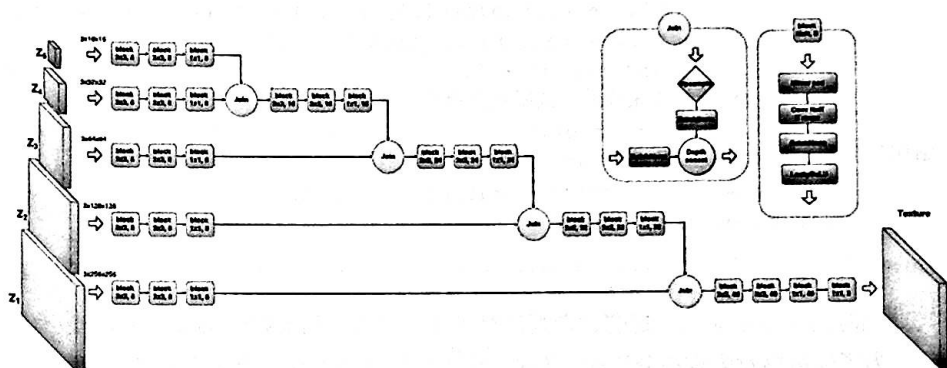


图 4-21 Texture Nets 生成器网络结构

Texture Nets 的实现代码如下：

```
def texture_net(input_image):
    conv2d = tflearn.conv_2d
    batch_norm = tflearn.batch_normalization
    relu = tf.nn.relu
    # 定义缩放尺度
    ratios = [16, 8, 4, 2, 1]
    # n_filter 为每一尺度的特征图数量
    n_filter = 8
    net = []

    for i in range(len(ratios)):
        # 首先使用 max_pooling 缩小图像
        net.append(tflearn.max_pool_2d(input_image, ratios[i],
        ratios[i]))
        # 构建每一个尺度下的 block_i_0, block_i_1, block_i_2
        for block in range(3):
            ksize = 1 if (block + 1) % 3 == 0 else 3
            net[i] = relu(batch_norm(conv2d(net[i], n_filter, ksize)))
        if i != 0:
            # 与 net[i-1] 合并
```

```

upnet = batch_norm(net[i - 1])
downnet = batch_norm(net[i])
net[i] = tf.concat(3, [upnet, downnet])
# 构建每一个尺度下的 block_i_3, block_i_4, block_i_5
for block in range(3, 6):
    ksize = 1 if (block + 1) % 3 == 0 else 3
    net[i] = conv2d(net[i], n_filter * (i + 1), ksize)
    net[i] = relu(batch_norm(net[i]))
if i != len(ratios) - 1:
    # 通过上采样的方式提高分辨率
    net[i] = tflearn.upsample_2d(net[i], 2)
# 转换为 3 通道图像并输出
output = conv2d(net[len(ratios) - 1], 3, 1)
return output

```

风格滤镜网络每次都是针对一张风格图进行训练，也就是说风格图是固定的，但与此同时内容图不能是固定的，否则最终训练的结果就与原始的 *neural style* 没有区别。为了使风格对内容充分泛化，不被某张图的内容所限定，需要用多张图片来训练网络，使风格能适应任意图片的分布。训练数据可以从 ImageNet 或者其他图像数据集中随机选择。训练集大小其实并没有限制，因为原本训练集的目的就在于让模型能够尽可能多地适应多种图像分布，但在 Ulyanov 后续的研究论文 *Instance Normalization: The Missing Ingredient for Fast Stylization* 中发现，使用上万张图片的训练集和长时间的训练反而会造成效果下降。这里的效果指的是人看到的感受，而不是损失函数的收敛效果。在论文中提到，用 16 张图片训练出的模型要比上千张图片训练出的模型生成的风格效果更加明显，甚至最好看的效果是用少量图片训练并且要适时提前中止训练才得来的。在本书作者看来这似乎有些玄幻，似乎要想训练出一个好看的风格滤镜的话，运气的成分也很重要²²。这也许就是 Prisma 至今还无人能够完美复制的原因吧。

4.5 小结

卷积神经网络在深度学习中扮演了重要的角色，AlexNet 让人们意识到了深度学习的强大，GoogLeNet、ResNets 让计算机拥有了超过人类的识别能力。神经网络在

²² 欢迎读者在 GitHub 对文中观点进行探讨：<http://github.com/DeepVisionTeam/TensorFlowBook>。

加入了卷积层之后威力完全不可同日而语，不管是识别还是定位物体都游刃有余，甚至还能创造出精美的艺术作品。

CNNs 让计算机有了“看懂”世界的能力，为我们开辟了一片广阔的应用领域，也注定会在未来改变我们的生活。

4.6 参考资料

- [1] ILSVRC2016 竞赛官方网站: <http://image-net.org/challenges/LSVRC/2016/index>
- [2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.
- [3] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).
- [4] Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2015.
- [5] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).
- [6] Szegedy, Christian, et al. "Rethinking the inception architecture for computer vision." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.
- [7] Szegedy, Christian, et al. "Inception-v4, inception-resnet and the impact of residual connections on learning." arXiv preprint arXiv:1602.07261 (2016).
- [8] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.

- [9] Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. "A neural algorithm of artistic style." arXiv preprint arXiv:1508.06576 (2015).
- [10] Ulyanov, Dmitry, et al. "Texture networks: Feed-forward synthesis of textures and stylized images." Int. Conf. on Machine Learning (ICML). 2016.
- [11] Johnson, Justin, Alexandre Alahi, and Li Fei-Fei. "Perceptual losses for real-time style transfer and super-resolution." European Conference on Computer Vision. Springer International Publishing, 2016.
- [12] Ulyanov, Dmitry, Andrea Vedaldi, and Victor Lempitsky. "Instance Normalization: The Missing Ingredient for Fast Stylization." arXiv preprint arXiv:1607.08022 (2016).

5

RNN “能说会道”

深度学习技术已在图像识别领域取得了令人瞩目的成绩。因为图像是二维的感知数据，深度 CNNs 神经网络非常适合被用于提取图像特征。实践表明，深度 CNNs 神经网络已经成为当前世界图像识别竞赛的主流方法。

自然语言处理（Natural Language Processing, NLP）同样是人工智能领域的研究热点。语言包括语音和文字两大部分。但从概念上讲自然语言处理只包括语言文字的处理。它是指实现人与计算机之间用自然语言进行有效通信的各种理论和方法。与语音信息相比，文字信息能够更好地反映语言的内在关联逻辑。与图像数据不同，语言文字数据是一维的字符序列。显然，深度 CNNs 神经网络不再适用于这种类型的数据。对此，人们提出了 RNN（Recurrent Neural Networks）循环神经网络来提取自然语言文字的序列信息，并用来建立数学模型解决相应问题。

自然语言处理在人工智能领域涉及的问题相当广泛，包括机器翻译、智能对话、文字语义理解等方面。目前深度学习技术已在机器翻译、智能对话等问题方面取得了非常不错的效果。例如 Google 翻译、iPhone 上的聊天机器人等。但在文字语义理解问题方面，仍有很多关键问题需要解决。著名的“图灵测试”是一种采用文字问答的方式来测试机器是否具有语义理解智能的专项测试。虽然，该项测试表明目前的智能算法能够让计算机具备一定的智能文字处理应答能力。但这还远远不能让计算机像人

类一样熟练地运用语言。

语言和文字是人类感知世界最重要的途径。它们更是人们记录思想、学习交流、传承文明的重要媒介。从语言学角度来说，“听说读写”四个方面是衡量一个人掌握一门语言的四个方面。对应的，若使计算机像人类一样具备语言的运用能力，同样应考量“听说读写”这四个方面。对此，人工智能领域发展出了如下四个最主要的研究方向。

- 听：语音识别（Speech To Text, STT），实现从音频到文本转换。
- 说：语音合成（Text To Speech, TTS），实现从文本到音频转换。
- 读：文本理解（Text Understanding），实现文本到语义转换。
- 写：文本生成（Text Generation），实现语义到文本转换。

其中，“读”和“写”，也就是文本理解和文本生成两个方向，它们同属于自然语言处理的范畴。与“听”“说”相比，它们更注重语义概念及其内在逻辑。因为文本数据具有易保存、噪声小、自带标签等特性，使得它更适合被用于分析语义概念。相比之下，语音数据存储量大、噪声处理复杂、数据类别标记困难。这导致语音识别和语音合成技术更侧重于信号分类与还原。

本章将主要介绍循环神经网络的基本概念及其在自然语言处理领域中的多种应用，并通过实际的例子介绍使用 TensorFlow 实现循环神经网络的具体方法。

5.1 文本理解和文本生成问题

文本理解和文本生成，是自然语言处理（Natural Language Processing）领域中的两个典型问题。

文本理解又称为文本语义分析。它是指基于词法、语法等信息，让计算机自动从文本中挖掘出有用的信息，并帮助人们理解文本内容的智能算法。典型的应用包括文本分类、情感分析、自动文摘等。

文本生成比较容易理解，就是让计算机自动输出符合人类语言习惯的文本。典型的应用场景如机器翻译、对话机器人，甚至“人工智能写作”等。

文本理解示例 1：新闻自动分类（文本分类）

对于媒体和门户网站来说，将每日的新闻按政治、经济、科技、娱乐等类别归档整理，是一项必不可少的工作。新闻自动分类算法，使用文本分类的技术，可以自动为每一篇新闻文档预测其所属类别，如图 5-1 所示。

- 财经 | 楼市整顿风暴来袭：北京有些小区房源全部下架
- 肖捷接棒楼继伟：中国财税改革已经进入深水区
- 漩涡中的河南首富：被下属妻子举报迫害高管 超生5子女
- 股票 | 保监会约谈恒大人寿负责人：不支持险资短炒股票
- 调控威力显现经理人信心走低 业内人士称房价难降
- 科技 | 乐视融资超500亿仍诉苦 危机解药：做实生态圈
- 抛光处理受欢迎？苹果可能增加“亮白色”iPhone 7
- 马斯克：未来人类都不上班 GIF：猫咪被打得无还手之力
- 汽车 | 哈弗新中型SUV曝光 夫妻拍涉违章警车被按下跪
- 时尚 | 5岁安吉24K纯爷们 范冰冰斗篷上身超拉风

图 5-1 媒体对新闻的分类示例

文本理解示例 2：用户评论分析（情感分析）

电商商家为了优化用户体验和商业收益，需要对用户评论进行分析和统计，如图 5-2 所示。通过文本理解技术，可以自动分析每一条评论分别反映了用户的正面情绪或是负面情绪，通过综合多个用户评论的情感分析结果，可以得到商品的用户偏好，为商业优化提出决策数据。甚至可以通过分析用户对产品、物流、服务中的某些方面提出的疑问，进行定向预警公关，将负面影响降低到最小的范围。

还不错，有点贵了，味道还有点。 今天	颜色分类：黑色 尺码：175(XL) 是否加绒：否	1***8 (匿名)
不好 今天	颜色分类：深灰色 尺码：175(XL) 是否加绒：否	6***y (匿名)
质量还可以吧 今天	颜色分类：军青 尺码：175(XL) 是否加绒：否	r***儿 (匿名)
不错啊，质量不错还很便宜 今天	颜色分类：军青 尺码：180(XXL) 是否加绒：否	m***5 (匿名)

图 5-2 用户评论分析（情感分析）

文本生成示例：机器翻译

随着全球化的发展，世界各地跨区域的交流与合作越来越频繁，多语言翻译的需求无处不在。人类翻译成本高昂，而且效率较低。相对而言，使用计算机进行自动机器翻译是一种不错的选择，如图 5-3 所示。



图 5-3 机器翻译

通过上述几个示例场景可以发现，文本数据是要研究和处理的对象。文本可以是一个句子、一个段落或者是一篇文章，从计算机表示的角度来说，文本就是由字和词组成的序列。文本数据与图像数据最大的区别在于，文本数据更强调文字序列中前后元素之间的相互影响。要预测句子中的下一个词，一般都要通过句子中的前一个或几个词进行推断，因为前后单词并不是独立的。要理解一句话也是相同的道理，相同的一组字词经过不同顺序排列得到的句子，可能在语义上截然不同，比如“不怕辣”，“怕不辣”，“辣不怕”和“怕辣不”就显然是不同的意思。文本数据的另一大特点就是文本序列属于变长数据。以新闻文本为例，一则新闻可以是十几个字的快讯，也可以是正常的文章报道，还有可能是长篇连载系列专题。对于图像数据来说，即使原始数据大小不同，还可以采取缩放或者裁剪的方式，在不影响语义的前提下统一数据的大小。但对于文本数据来说，并不存在这样的转换方案，任何裁剪都可能会导致语义信息的丢失。因此，针对文本处理所设计的模型，需要能够捕捉和利用数据的顺序，并且能处理变长的数据。

传统的前馈神经网络（Feed-forward Neural Network, FNN）并不适合于文本处理问题。首先，文本数据是变长的，而传统神经网络的输入和输出都是固定长度的。如果输入层有 5 个节点，那就只能接收 5 个元素的输入。其次，即使通过截取、填充等手段将文本数据全部都归一化到固定长度后，由于传统前馈神经网络假设所有输入节点之间是独立的，所以依然不能准确提取一句话中前后单词之间的局部语义关系，而是尝试提取单词在整个语句中的整体语义关系的期望均值。FNN 网络的输入特征内部是没有任何联系的，因此并不适用于自然语言处理。

图 5-4 展示了使用经典的前馈神经网络解决序列数据分析问题的流程。在图中，输入层节点为 $[x_1, x_2, \dots, x_m]$ ，经过若干个隐含层的计算，最终到达输出层并得到预测结果 y 。对于输入的序列数据 (w_1, w_2, \dots, w_m) ，每一个元素对应到一个输入节点的位置。从图中可以看出任意两个输入数据 w_i 和 w_j ，它们作用于最终的输出结果是由不同的模型参数决定的。整个 w_i 和 w_j 两者之间并无任何关联。由于输入层与隐含层之间是全连接，故可以认为对于中间层来说，每一个输入节点都是独立的、平等的， w_1 与 w_2 、 w_2 与 w_3 之间的依赖关系在网络结构中完全没有体现，也就更谈不上学习了。

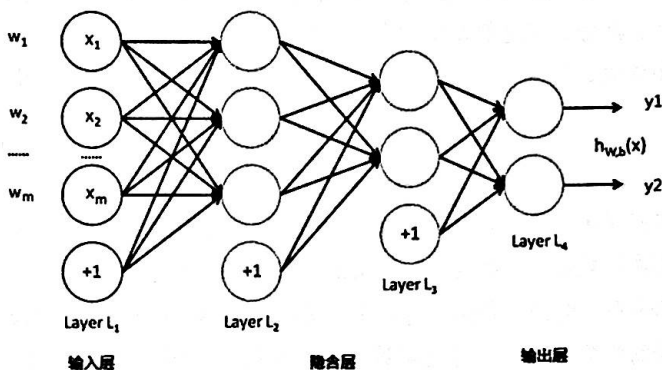


图 5-4 传统的前馈神经网络

曾有人尝试用卷积神经网络 CNN 的方法来处理文本数据，也就是说，对一维的序列数组进行卷积操作。实际上，卷积的输出只是输入序列中很少几个邻近元素的函数，同样并不能表达文本中的语序关系。

面对这样一种情况，循环神经网络给出了另外一种解决思路。在循环神经网络中，每一个输出元素的生成都基于同一个网络，这样就可以简单地输出变长结果。输出序列的每个元素都是其之前位置的输出元素所组成的函数，这就保持了序列元素的顺序依赖关系。5.2 节将详细介绍循环神经网络是如何处理文本数据的。

5.2 标准 RNN 模型

5.2.1 RNN 模型介绍

循环神经网络 (Recurrent Neural Networks, RNNs) 是专门用于处理序列数据的深度学习模型。正如 5.11 节所介绍的, 文本处理问题上最需要解决的就是序列语序和变长数据的问题。

在理解 RNN 之前, 不妨先考虑一下人类是如何分析一系列事件之间的联系。最基本的方式, 就是记住之前发生的事, 将事件有条理地存储在大脑的记忆当中。在判断下一个事件时, 先回忆起之前发生的事, 并根据思维模式推理出最合理的情况。

RNN 网络就是在传统神经网络的基础上加入了“记忆”的成分。对于 RNN 模型来说, 序列 (x_1, x_2, \dots, x_m) 被看作一系列随着时间步长 (time step) 递进的事件序列。这里的时间步长并不是真实世界中所指的时间, 而是指序列中的位置。如果用传统神经网络对序列进行处理, 网络会对 (x_1, x_2, \dots, x_m) 这些输入元素分别计算, 得到对应的 (o_1, o_2, \dots, o_m) 一系列输出, 但 o_1 与 o_2 并不无关联。RNN 之所以叫做“recurrent”, 是因为它在对当前时刻状态的计算还依赖于前一步 (前一个位置) 的计算结果, 也就是说, o_2 的计算实际上是同时基于 x_2 和 o_1 的。对于时刻 t 来说, 上一个时刻的状态表示为 h_{t-1} , 可以说 h_{t-1} 编码记录了 x_t 的前缀序列数据 $(x_1, x_2, \dots, x_{t-1})$ 的信息; 当对 t 时刻的输入 x_t 进行计算时, 循环神经网络综合当前时刻输入 x_t 与前一个时刻的“记忆” h_{t-1} , 一起得到 t 时刻的状态 h_t 。

RNN 的基本形式如图 5-5 所示。图中, x 是输入序列向量, h 代表网络隐层的状态, o 代表输出向量。输入与隐层之间通过参数矩阵 U 连接, 不同时刻的隐层之间以参数矩阵 W 连接, 隐层与输出层之间以参数矩阵 V 连接。

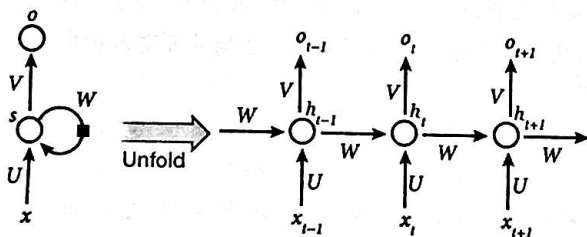


图 5-5 循环神经网络的折叠形式和展开形式

图 5-5 的右图为 RNN 的展开形式，左图为简化表示后的折叠形式。左图中的黑色方块，描述了一个延迟连接，从上一个时刻的隐含状态 h_{t-1} 到当前时刻隐层状态 h_t 之间的连接。

用公式来表示，RNN 的前向传播可以表示为：

$$a_t = b + Wh_{t-1} + Ux_t$$

$$h_t = \tanh(a_t)$$

$$o_t = c + Vh_t$$

$$\hat{y}_t = \text{softmax}(o_t)$$

其中， x_t 为 t 时刻的输入，表示 h_t 为 t 时刻的隐层状态表示， o_t 表示 t 时刻的输出表示， \hat{y}_t 为经过归一化后的预测概率。

值得一提的是，参数共享（Parameter sharing）在 RNN 中也起到了至关重要的作用。从图 5-5 和上面的公式中可以注意到，对于每一个时刻，参数矩阵 U 、 W 、 V 并没有变化，而是使用同一组参数。参数共享的意义在于，在一段文本中，部分重要的信息可能出现在序列的任何位置上，甚至同时出现在多个位置上。比如，“小明热爱深度学习”和“深度学习是小明的最爱”这两句话表达的是类似的意思，如果让机器学习模型来理解其中的含义，判断小明喜欢的学科，那“深度学习”不管出现在文本的开始或是结尾，都应该被认为是最具有相关性的信息。也正是由于参数共享的作用，RNN 可以处理任意长度的序列，只要序列的元素按顺序一个一个地传入网络，RNN 网络就像管道一样逐个处理。

另外，需要特别说明的是，关于 RNN 这个缩写，某些情况下会出现混淆。循环神经网络（Recurrent Neural Network）和递归神经网络（Recursive Neural Network）有着共同的缩写 RNN，而本章讨论的是循环神经网络，也是各种文献中多数情况下 RNN 的含义。

递归神经网络的网络结构与循环神经网络有很大不同。递归神经网络通过构建参数共享的树状结构处理序列数据，其结构如图 5-6 所示。

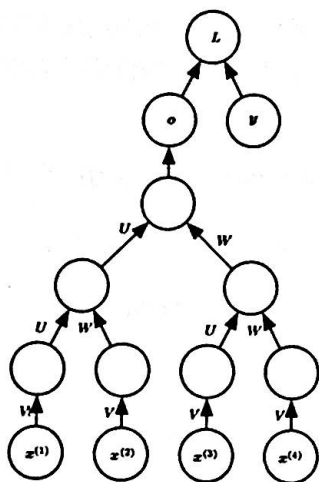


图 5-6 递归神经网络的结构

在考虑具体的输入数据的情况下，递归神经网络一般采取固定的平衡二叉树结构的方式构造。“递归”的含义在于树状网络结构中的重复子结构模块之间实施了参数共享。如果说循环神经网络是在时间维度上进行参数共享，那么递归神经网络是在空间维度上进行参数共享。

5.2.2 BPTT 算法

当使用传统的神经网络进行监督学习时，模型的训练本质上就是通过最小化损失函数，以确定模型参数的取值。对于传统的神经网络来说，一般使用经典的反向传播（Back-propagation, BP）算法进行参数更新。BP 算法的每次迭代包含两个阶段，即前向传播（forward）阶段和反向传播（backward）阶段。在前向传播阶段，模型完成从输入节点到输出节点的计算过程，直到最终获得当前模型的损失函数值为止。反向传播阶段则从输出节点到输入节点计算每个参数的梯度值，并根据学习率进行更新。然而，对于循环神经网络来说，由于网络中“循环”的存在，权值参数 W 、 U 、 V 矩阵都在网络中多个层的计算中共享，使得普通的 BP 算法不能直接应用在 RNN 模型的训练中。

BPTT（Back-Propagation Through Time）算法是 BP 算法在 RNN 结构上的一种变体形式，采取在 RNN 模型的展开式上进行梯度计算。BPTT 在循环神经网络中的反

向传播过程如下：

首先为了简化推导，可以将上一小节描述的 RNN 节点的公式表示中的偏置参数忽略，使用下面这个更简单的表示：

$$h_t = \tanh(Ux_t + Wh_{t-1})$$

$$\hat{y}_t = \text{softmax}(Vh_t)$$

其中 \hat{y}_t 为 t 时刻输出的概率结果， h_t 为 t 时刻的状态， W 、 U 、 V 为参数。反向传播的目的是根据前向传播得到的代价函数值 L ，计算参数矩阵 W 、 U 、 V 中各个值的梯度，以便应用随机梯度下降算法求得最合适的参数值。

此时循环神经网络的展开形式可以如图 5-7 所示。

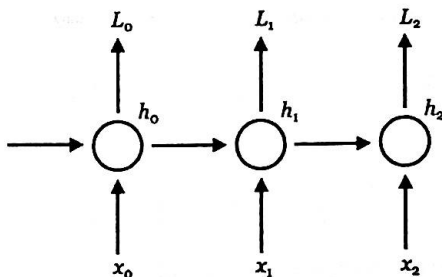


图 5-7 展开形式的 RNN

以图 5-7 中的 L_2 为例与计算于参数 W 的梯度，根据链式法则有：

$$\frac{\partial L_2}{\partial W} = \frac{\partial L_2}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial h_2} \cdot \frac{\partial h_2}{\partial W}$$

然而在其中，由于 $h_2 = \tanh(Ux_2 + Wh_1)$ ， h_2 对 h_1 有依赖， h_1 又对 h_0 有依赖，所以需要继续使用链式法则，得出：

$$\frac{\partial L_2}{\partial W} = \sum_{k=0}^2 \frac{\partial L_2}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

因为在循环神经网络每一个时刻的计算中都使用参数 W ，所以此处会将从 $t = 0$ 时刻到 $t = 2$ 时刻之间所产生的梯度全部进行累加。同理，参数 U 和 V 的梯度计算方

式与 W 相似。

其实从计算过程上来说, BPTT 与标准反向传播算法基本相同。最主要的差异就是由于在 RNN 网络中, 每一层计算都共享参数变量 W , 所以在计算梯度的时候, 每一层所得到的 W 的梯度都需要累加在一起, 以保证每一层计算所得的误差都得到一定程度的矫正。而在传统的神经网络中, 每一层计算的参数都相互独立, 隐层之间并没有共享参数, 所以对梯度的学习也都是独立的。

5.2.3 灵活的 RNN 结构

针对不同的应用场景, RNN 网络结构可以有多种灵活的形式。从输入和输出是否为变长的角度考虑, 可以分为: one to one、one to many、many to one、many to many、many to many(sync), 如图 5-8 所示。

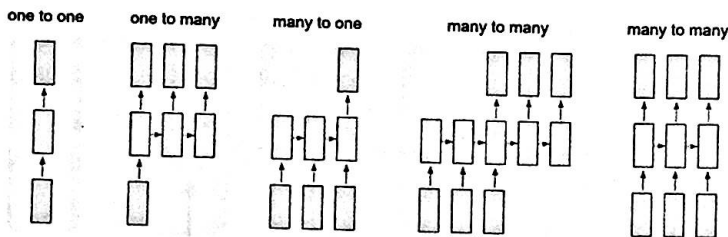


图 5-8 从输入/输出是否变长的角度, 循环神经网络结构的五种形式

图 5-8 中, 每个方框都代表了一个向量, 底层的方框代表输入, 顶层的方框代表输出, 中间方框代表模型的隐层的 RNN 状态向量。同时, 每个箭头都代表施加在向量上的计算函数。

one to one 的意思是单输入单输出网络。这里的单输入, 并非表示网络的输入向量长度为 1, 而是指数据的长度是确定的。比如输入数据可以是一个固定类型的数, 可以是一个固定长度的向量, 或是一个固定大小的图片。同理, 模型输出规模也是确定的。传统神经网络和 CNN 都可以理解为是这种形式的结构, RNN 并不在其中。在这一类别中, 典型的应用场景比如传统的文本分类算法。对于给定文章 D , 基于人工设计的特征抽取算法可以得到文章的特征向量 (f_1, f_2, \dots, f_m) , 将特征向量输入神经网络, 计算出文章属于 c 个分类中每个类别的概率 (p_1, p_2, \dots, p_c) 。在这个例子中, 虽然输入的特征数 m 和输出的分类数 c 都大于 1, 但它们都是确定的, 因此, 网络结构记

为 one to one。

one to many 表示定长输入变长输出的网络结构,如图 5-9 所示。以单词释义问题为例,输入的是单个的英文单词,比如“he”,输出将是一段单词序列“that male one who is neither speaker nor hearer”。

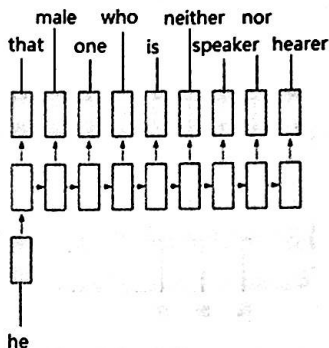


图 5-9 one to many 结构, 举例: 英文释义问题

many to one 是指输入为变长序列、输出为固定长度的模型,如图 5-10 所示。文本情感分析问题就是这一类型中典型的场景,网络模型要根据一段文本,给出是否为积极情绪或者消极情绪的概率判断。

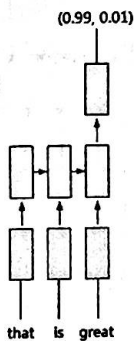


图 5-10 many to one 结构, 举例: 情感分析问题

many to many 有两种形式。第一种属于 Encoder-Decoder 框架,它的特点是输入序列数据经过编码器网络得到内部表示后,再基于这个表示通过解码器网络生成新的

序列数据。输入和输出都是变长数据，而且两者的长度可以不同。这一类型的典型场景是机器翻译问题。如图 5-11 所示，使用 RNN 进行中英翻译，网络输入为一段中文文本，输出为输入文本对应的英文翻译结果。

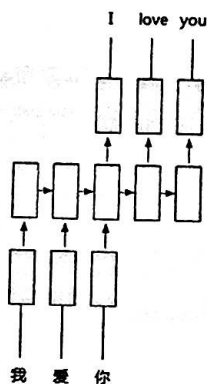


图 5-11 many to many 结构，举例：机器翻译问题

many to many 的第二种结构是输入和输出的元素完全一一对应，虽然输入数据和输出数据的规模都是不定的，但对于每一个输入，都有对应输出。也就是说，输入数据和输出数据长度是一致的。因为输入输出之间是相互对应的，也可称之为同步版本的多输入多输出。文本序列标注问题就是这样一类问题。如图 5-12 所示，可以使用 RNN 对给定文本中的每一个单词进行词性标注。例子中的“我”、“爱”、“你”三个词语的词性分别是“r” (pronoun, 代词)、“v” (verb, 动词)、“r” (pronoun, 代词)。

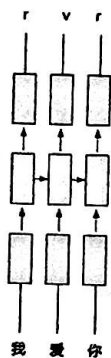


图 5-12 many to many (sync) 结构，举例：文本序列标注问题

通过这些不同的形式和场景可以看到，RNN 有着比固定输入输出的网络更加强大的能力和更多适用场景。

5.2.4 TensorFlow 实现正弦序列预测

如同对卷积操作的支持一样，TensorFlow 对 RNN 模型的基础结构 RNN 单元也提供了实现和封装，使用 TensorFlow 可以轻松地完成一个序列预测模型。在这一小节里，就通过实现正弦函数序列的预测示例，介绍 TensorFlow 实现 RNN 模型的方式。

为了方便演示，在本例中首先使用 numpy 构造出要使用的训练数据和测试数据：

```
import random
import numpy as np

def build_data(n):
    xs = []
    ys = []
    for i in range(2000):
        k = random.uniform(1, 50)

        # x[i] = sin(k + i) (i = 0, 1, ..., n-1)
        # y[i] = sin(k + n)
        x = [[np.sin(k + j)] for j in range(0, n)]
        y = [np.sin(k + n)]
        xs.append(x)
        ys.append(y)

    train_x = np.array(xs[0: 1500])
    train_y = np.array(ys[0: 1500])
    test_x = np.array(xs[1500:])
    test_y = np.array(ys[1500:])
    return (train_x, train_y, test_x, test_y)

# build data
(train_x, train_y, test_x, test_y) = build_data(length)
print(train_x.shape, train_y.shape, test_x.shape, test_y.shape)
```

在上面程序中一共构造了 2000 个序列数据样本,其中前 1500 个样本组成训练集,后 500 个样本组成测试集。

随后,使用 BasicRNNCell 构建 RNN 预测模型:

```
import tensorflow as tf
from tensorflow.contrib.rnn.python.ops import core_rnn
from tensorflow.contrib.rnn.python.ops import core_rnn_cell

length = 10
time_step_size = length
vector_size = 1
batch_size = 10
test_size = 10

X = tf.placeholder("float", [None, length, vector_size])
Y = tf.placeholder("float", [None, 1])

W = tf.Variable(tf.random_normal([10, 1], stddev=0.01))
B = tf.Variable(tf.random_normal([1], stddev=0.01))

def seq_predict_model(X, w, b, time_step_size, vector_size):
    # input X shape: [batch_size, time_step_size, vector_size]
    # transpose X to [time_step_size, batch_size, vector_size]
    X = tf.transpose(X, [1, 0, 2])
    # reshape X to [time_step_size * batch_size, vector_size]
    X = tf.reshape(X, [-1, vector_size])
    # split X, array[time_step_size], shape: [batch_size,
    vector_size]
    X = tf.split(X, time_step_size, 0)

    cell = core_rnn_cell.BasicRNNCell(num_units=10)
    initial_state = tf.zeros([batch_size, cell.state_size])
    outputs, _states = core_rnn.static_rnn(cell, X,
                                           initial_state=initial_state)

    # Linear activation
    return tf.matmul(outputs[-1], w) + b, cell.state_size

pred_y, _ = seq_predict_model(X, W, B, time_step_size, vector_size)
```

接着定义代价函数和优化算法：

```
loss = tf.square(tf.subtract(Y, pred_y))
train_op = tf.train.GradientDescentOptimizer(0.001).minimize(loss)
```

随后就可以使用最开始构造的正弦序列进行学习了：

```
with tf.Session() as sess:
    tf.global_variables_initializer().run()

    # train
    for i in range(50):
        # train
        for end in range(batch_size, len(train_x), batch_size):
            begin = end - batch_size
            x_value = train_x[begin: end]
            y_value = train_y[begin: end]
            sess.run(train_op, feed_dict={X: x_value, Y: y_value})

            # randomly select validation set from test set
            test_indices = np.arange(len(test_x))
            np.random.shuffle(test_indices)
            test_indices = test_indices[0: test_size]
            x_value = test_x[test_indices]
            y_value = test_y[test_indices]

            # eval in validation set
            val_loss = np.mean(sess.run(loss,
                                         feed_dict={X: x_value, Y: y_value}))

        print('Run %s' % i, val_loss)
```

通过上面代码可以看到，使用 TensorFlow 的 BasicRNNCell 和 static_rnn 接口，可以非常容易地构造出一个包含 10 次循环的 RNN 网络。通过 TensorBoard 可以清晰地看出循环神经网络展开式的结构，如图 5-13 所示。

虽然这只是一个非常简单的小例子，但从中可以看出使用 TensorFlow 构造 RNN 网络模型的基本用法。以此为基础，构造更复杂的模型也就不再是困难的事。

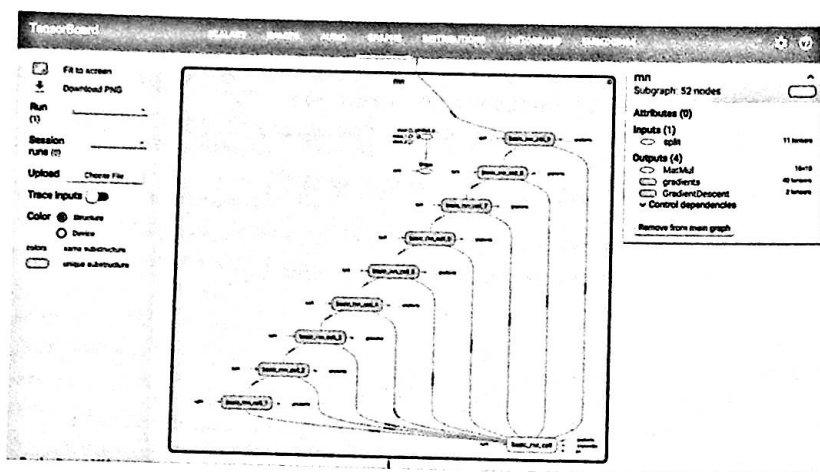


图 5-13 通过 TensorBoard 展示示例代码所构造的 RNN 网络结构

5.3 LSTM 模型

RNN 是在有序的数据上进行学习的，为了记住这些数据，RNN 会像人一样对之前发生事件产生记忆。但是 RNN 的结构决定了它只能对距离比较近的時刻的记忆更加强，而对距离久远的事件记得并不清楚。

以文本理解为例，首先考虑一段文本：“下面我们来介绍使用 TensorFlow 实现 ‘Hello World’ 的程序，首先 TensorFlow 是一个深度学习框架，它使用计算图作为计算过程的抽象。Blablabla.....这样，第一个用 TensorFlow 实现的____就完成啦！”。要预测文本结尾空白处的词，根据文本开头的信息可以推断出应该是“程序”。但如果用 RNN 模型来预测，文本开头的“程序”一词极其相关信息要经过长途跋涉传递到文本末尾处。虽然理论上 RNN 能够处理这种长期依赖（Long-Term Dependency）的问题，但从实践来看，普通的 RNN 模型并不能学习到远距离的知识。

5.3.1 长期依赖的难题

长期依赖对于文本理解是不可回避的问题，但普通 RNN 结构并不能很好地处理这个问题。普通 RNN 网络结构“健忘”的原因，可以从数学上稍作分析。通过上一节的介绍可以知道，RNN 结构本质上是很多层相同非线性函数的嵌套形式。更具体

来说,如果暂时忽略掉激活函数和输入向量 x , RNN 节点的状态 h_t 可以表示成:

$$h_t = Wh_{t-1} = W^2h_{t-2} = \dots = W^th_0$$

若对参数矩阵 W 进行特征分解

$$W = Q\Lambda Q^{-1}$$

其中,矩阵 Q 的第 i 列为 W 的特征向量 q_i 。 Λ 是对角矩阵,其对角线上的元素为对应的特征值。

通过特征分解,可以得到 RNN 节点状态 h_t 的更简化表示:

$$h_t = Q^{-1}\Lambda Q h_{t-1} = Q^{-1}\Lambda^{t-1} Q h_1 = Q^{-1}\Lambda^t Q h_0$$

可以看到,对于状态 h_t 的表示是与参数矩阵特征值的 t 次方相关的函数,而若特征值的取值小于 1,则最终结果会快速降为 0,而若特征值大于 1,则会快速趋向于无穷大。

由于参数矩阵 W 在每一次循环的计算中被共享,所以在状态传递过程中相当于 W 被乘了很多次, W^t 只可能趋向于 0 或无穷大,即消失或爆炸。这也就是说,对一个序列,任何前面发生的事件的记忆都会以指数级的速度被遗忘,最终导致长距离的信息很难在网络中传递。

5.3.2 LSTM 基本原理

LSTM 就是为了解决长期依赖问题而诞生的。长短期记忆网络 (Long Short Term Memory Networks, LSTM) 是循环神经网络的一个改进,由 Sepp Hochreiter 和 Jurgen Schmidhuber 于 1997 年首次提出。

如图 5-14 所示, LSTM 和普通 RNN 相比,最主要的改进就是多出了三个门控制器:输入门 (input gate)、输出门 (output gate) 和遗忘门 (forget gate)。三个门控制器的结构都相同,主要由 sigmoid 函数 (图 5-14 右图中的“ σ ”节点) 和点积操作 (图 5-14 右图中的“ \times ”节点) 构成。由于 sigmoid 函数的取值范围是 $[0, 1]$,所以门控制器描述了信息能够通过的比例, sigmoid 取值为 0 的时候表示没有信息能够通过,或者理解为将所有记忆全部遗忘。反之,取值为 1 时表示所有信息都能通过,完全保留

这一分支的记忆。

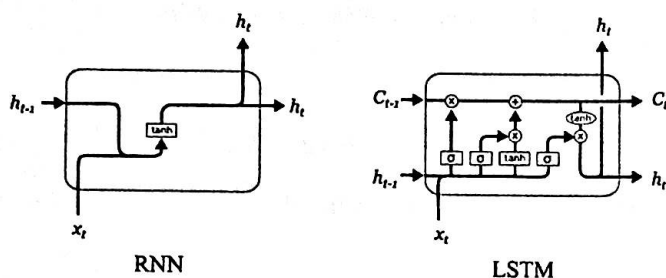


图 5-14 标准循环神经网络和 LSTM 模型对比，LSTM 模型增加了状态 C_t 和三个门结构

对于标准循环神经网络，每个时刻的状态都由当前时刻的输入与原有的记忆结合组成。但问题在于记忆容量是有限的，如前面介绍的，早期的记忆会呈指数级衰减。为了解决这一问题，LSTM 模型在原有的短期记忆单元 h_t 的基础上，增加一个记忆单元 C_t 来保持长期记忆。

具体来说，长期记忆单元 C_t 的更新如图 5-15 所示。

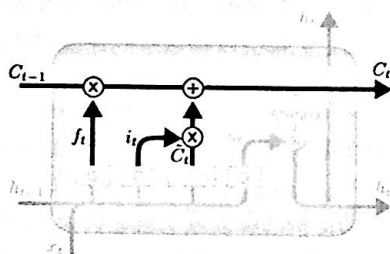


图 5-15 LSTM 长期记忆单元 C_t 的更新示意图

公式表示为：

$$C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t$$

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

其中, f_t 、 i_t 分别代表遗忘门和输入门, 在每一个时刻, 遗忘门会控制上一时刻记忆的遗忘程度, 输入门会控制新记忆 \tilde{C}_t 的写入长期记忆的程度。 f_t 、 i_t 和 \tilde{C}_t 都是与上一时刻的短期记忆 h_{t-1} 和当前时刻输入 x_t 相关的函数。并且 f_t 和 i_t 是 sigmoid 的函数, 所以取值范围为 $[0, 1]$, \tilde{C}_t 为 \tanh 的函数, 取值范围为 $[-1, 1]$ 。

另一方面, 对于短期记忆 h_t 的更新如图 5-16 所示:

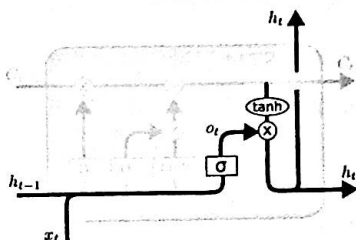


图 5-16 LSTM 短期记忆单元 h_t 的更新示意图

公式表示为:

$$h_t = o_t \times \tanh(C_t)$$

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o)$$

其中, o_t 表示输出门, 它控制着短期记忆如何受长期记忆影响。

从网络结构设计角度来说, LSTM 对标准 RNN 的改进主要体现在通过门控制器增加了对不同时刻记忆的权重控制, 以及加入跨层连接削减梯度消失问题的影响。图 5-17 展示了标准 RNN 与 LSTM 的节点级联及其抽象表示。从中可以看到, LSTM 实际上在原有结构上增加了线性连接, 而不再是单纯的非线性连接叠加, 这样就能使长期信息更好地传播。这与深度残差网络 ResNets 通过增加跨层连接 (Skip Connection) 来消除梯度消失问题的影响在设计上有异曲同工之妙。也由此可见, 在深度学习领域的模型改进方法上, CNN 与 RNN 可以说是殊途同归。

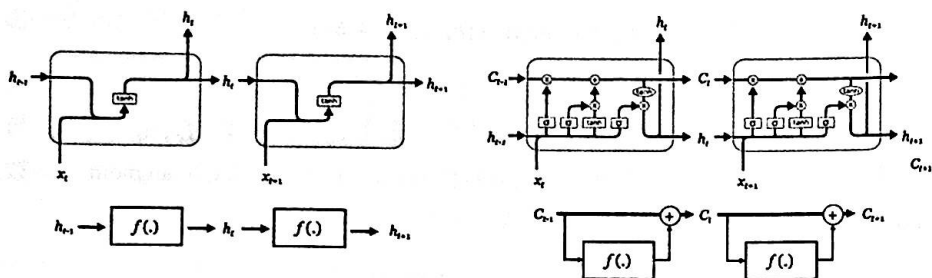


图 5-17 标准 RNN 与 LSTM 的节点级联及其抽象表示

通过上面的介绍不难看出，LSTM 通过门控制器和新的记忆单元，在 RNN 原有的短期记忆之上保留了长期记忆。如果一个事件非常重要，则输入门就按重要程度将短期记忆合并进长期记忆，或者通过遗忘门忘记部分长期记忆，按比例替换为现在的新记忆。而在最后，输出门会基于长期记忆和短期记忆综合判断到底应该有什么样的输出。基于这样的控制机制，LSTM 对于长序列的理解分析能力大幅提高，在多种应用中都取得了非常好的效果。

5.3.3 TensorFlow 构建 LSTM 模型

由于 LSTM 在研究和应用中出色的表现，已经逐渐成为处理序列数据的常用方法。TensorFlow 对于 LSTM 也有很好的支持。这里还以正弦序列预测为例，LSTM 版本的模型实现仅需要换上 LSTM 单元即可，代码如下。

```
def seq_predict_model(X, w, b, time_step_size, vector_size):
    # input X shape: [batch_size, time_step_size, vector_size]
    # transpose X to [time_step_size, batch_size, vector_size]
    X = tf.transpose(X, [1, 0, 2])
    # reshape X to [time_step_size * batch_size, vector_size]
    X = tf.reshape(X, [-1, vector_size])
    # split X, array[time_step_size], shape: [batch_size, vector_size]
    X = tf.split(X, time_step_size, 0)

    # LSTM model with state_size = 10
    cell = core_rnn_cell.BasicLSTMCell(num_units=10,
                                       forget_bias=1.0,
                                       state_is_tuple=True)
    outputs, _states = core_rnn.static_rnn(cell, X, dtype=tf.float32)
```

```
# Linear activation
return tf.matmul(outputs[-1], w) + b, cell.state_size
```

同样通过 TensorBoard 可以看到模型内部结构的展开式, 如图 5-18 所示:

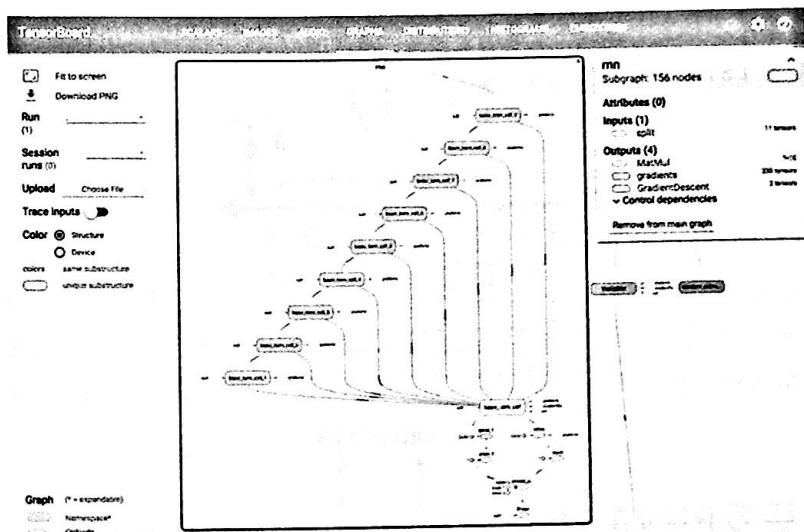


图 5-18 通过 TensorBoard 展示示例代码所构造的 LSTM 网络结构

值得注意的是, TensorFlow 中对于 LSTM 的实现有两种基本算子, 一个是基础的 BasicLSTMCell, 其中实现了基础的 LSTM 层。另一个是完整版的 LSTMCell, 它在原始 LSTM 的基础上, 加入了对 peephole 连接的支持。加入 peephole 连接的 LSTM 节点如图 5-19 所示。peephole 连接的作用是在输入、遗忘、输出控制器处理之前, 加入了对长期记忆的依赖。

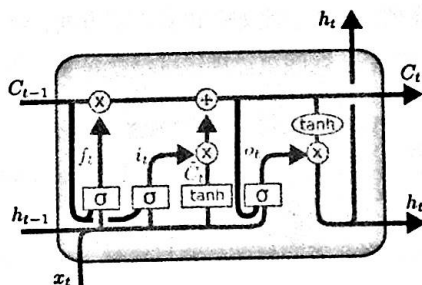


图 5-19 加入 peephole 连接的 LSTM 变体

TensorFlow 中原生支持的另一个 LSTM 的变体是 GRU (Gated Recurrent Unit) 模型, 对应代码的类名为 GRUCell。GRU 单元如图 5-20 所示, 主要特点是简化了节点状态和门单元设计, 从 LSTM 的遗忘门、输入门和输出门, 减少为重置门 (reset gate) 和更新门 (update gate), 从而显著减少了模型的参数个数。在实践中, GRU 模型效果跟 LSTM 不相上下, 但由于模型参数更少, 在计算性能方面的优势较为明显, 因此 GRU 是 LSTM 重要替代选择。

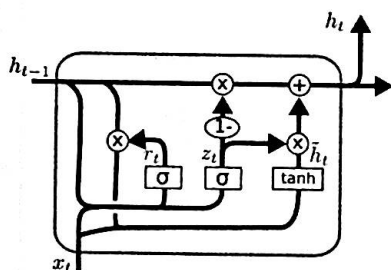


图 5-20 GRU 模型单元

5.4 更多 RNN 的变体

在标准循环神经网络的基础上, 结合不同应用场景和改进需求, 循环神经网络陆续出现了新的变种。其中比较著名的有双向循环神经网络 (Bidirectional RNNs) 和深度循环神经网络 (Deep Recurrent Networks)。

普通 RNN 或 LSTM 都是要构建一个具有“因果关系”的结构, 即当前时刻的输出依赖于之前所发生的事件。然而对于某些应用来说, 对当前时刻的判断需要依赖对于后续元素, 甚至整个序列的理解。比如在语音识别方面, 就需要根据后续序列识别的结果, 确定当前音节的归属。双向循环神经网络就是解决类似问题的方法, 其核心思想是, t 时刻的状态不单依赖 $t-1$ 时刻的状态, 还考虑从 $t+1$ 时刻到 t 时刻的依赖关系。网络结构如图 5-21 所示。

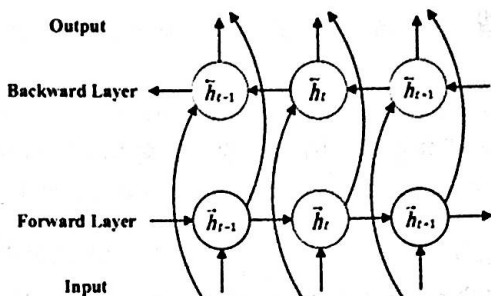


图 5-21 双向循环神经网络

在 TensorFlow 中,可以通过 `static_bidirectional_rnn` 方法构建双向循环神经网络。

RNN 的另一个主要变种,深度循环网络的主要思路是将每个时刻的处理过程加深,将多个节点堆叠在一个时刻的计算中,以加深非线性变换的复杂程度。研究表明这种结构会进一步改进 RNN 网络的效果。深层(多层)循环神经网络如图 5-22 所示。

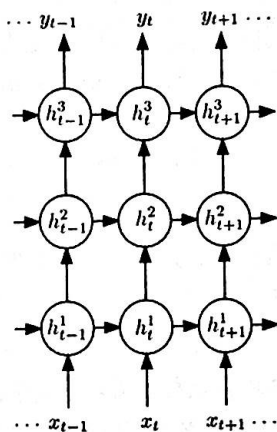


图 5-22 深度(多层)循环神经网络

在 TensorFlow 中,对这些前沿的模型结构都有比较完整的支持。双向循环神经网络可以通过 `static_bidirectional_rnn` 方法构建,深度循环网络也有对应的实现 `MultiRNNCell`。除此之外,从 TensorFlow 中还能看到不少业界最新的循环神经网络的变种,比如在中英文翻译和维基百科字符预测问题上都取得突出成绩的 Grid LSTM 等。可见,在 TensorFlow 的帮助下,应用也可以紧跟学术前沿。

5.5 语言模型

对于很多自然语言处理领域的问题，比如机器翻译，除了要确定预测结果中的字词集合以外，还有一个非常重要的方面就是要评估文本序列是否符合人类使用的习惯。通俗地说，就是要判断文本是否通顺、自然，甚至在翻译问题上，“信”、“达”、“雅”是一种更加高级的要求。语言模型就是用于评估文本符合语言使用习惯程度的模型。

要让机器来评估文本是否符合人类的使用习惯，一种方式是通过语言学方面的研究，制定出人类语言的范式，比如说陈述句是由主谓宾语构成的、定语修饰语需要加在名词前面等等。然而，所有人类语言的共同特点就是字词组合具有非常大的灵活性，同一个语义可以有多种表达方式，甚至许多二义性的语言带来了幽默的成分，是喜剧的重要组成部分。这种灵活性对规则的制定带来了巨大的难题。

以统计学为基础的统计语言模型是目前评估文本质量的主要方法。统计语言模型是基于预先收集的大规模的语料数据，以真实的人类语言为标准，预测文本序列在语料库中可能出现的概率，并以此概率作为评判文本是否“合法”的指标。从发展历史上看，统计语言模型大致经历了三个主要阶段：N-Gram 语言模型，神经网络语言模型，循环神经网络语言模型。在本节中将依次介绍这三种常用的模型。

5.5.1 N-Gram 语言模型

N-Gram 模型，也称为 N 元模型，是自然语言处理中常用的语言模型。该模型首先基于马尔科夫假设，即：假设在一段文本中，第 n 个单词出现的概率只与前面有限的 $n-1$ 个词相关，而与其他词都不相关。基于这样一种假设，可以评估文本中每一个词出现的概率，那么这整段文本的合法程度可以通过统计每 n 个词在语料库中出现的可能性来评估，整段文本的概率就是各个词出现概率的乘积。

假定 S 表示一个有意义的句子，它由一串特定顺序排列的词 (w_1, w_2, \dots, w_m) 组成， m 表示句子的长度，即单词的个数。计算 S 在整个语料库中出现的可能性 $P(S)$ ，或表示成 $p(w_1, w_2, \dots, w_m)$ ，可以根据链式法则，分解为：

$$\begin{aligned} P(S) &= p(w_1, w_2, \dots, w_m) \\ &= p(w_1) \times p(w_2|w_1) \times p(w_3|w_1, w_2) \times \dots \times p(w_m|w_1, w_2, \dots, w_{m-1}) \end{aligned}$$

基于马尔可夫假设：每一个词都只与最多前 $n-1$ 个词有关。也就是说：

$$p(w_i | w_1, w_2, \dots, w_{i-1}) = \begin{cases} p(w_i | w_1, w_2, \dots, w_{i-1}), & i < n \\ p(w_i | w_{i-(n-1)}, w_{i-(n-2)}, \dots, w_{i-1}), & i \geq n \end{cases}$$

当整段文本长度 m 远远大于 n 时的概率计算可以简化为：

$$\begin{aligned} P(S) &= p(w_1, w_2, \dots, w_m) = \prod_{i=1}^m p(w_i | w_1, w_2, \dots, w_{i-1}) \\ &\approx \prod_{i=n}^m p(w_i | w_{i-(n-1)}, w_{i-(n-2)}, \dots, w_{i-1}) \end{aligned}$$

对于每一个词出现的条件概率，可以通过在语料库中统计计数的方式得出。其中， $\text{count}(w_{i-(n-1)}, w_{i-(n-2)}, \dots, w_{i-1}, w_i)$ 表示字符串 $w_{i-(n-1)}, w_{i-(n-2)}, \dots, w_{i-1}, w_i$ 出现在语料库中的次数。

$$p(w_i | w_{i-(n-1)}, w_{i-(n-2)}, \dots, w_{i-1}) = \begin{cases} \frac{\text{count}(w_{i-(n-1)}, w_{i-(n-2)}, \dots, w_{i-1}, w_i)}{\text{count}(w_{i-(n-1)}, w_{i-(n-2)}, \dots, w_{i-1})}, & n > 1 \\ \frac{\text{count}(w_i)}{\sum_j \text{count}(w_j)}, & n = 1 \end{cases}$$

根据上述模型定义可以看出，当 n 越小时，模型只考虑邻近词语之间的关系。尤其是对于 $n=1$ 的特殊情况，被称之为 unigram，此时对于每一个词的概率评估实际上与文本的上下文无关，仅与当前词语在语料库中出现的概率有关。但人们不会以一个词一个词的方式交流，而是要以词组成句子和段落，所以要预测一个词是否出现，需要考虑上下文中的更多词，即增大 n 的取值，以捕捉更多的有用信息。

然而，另一方面，当 n 越大时，虽然模型会考虑更长距离的上下文之间的关联关系，但随着 n 的取值增大，语言模型的参数越多，将导致参数空间过大到无法估算。若词表集合为 V ，其中的单词数量为 $|V|$ ，则由这些词组成的 N 元组合的数目为 $|V|^n$ ，也就是说，组合数会随着 n 的增大呈指数级增长。同时，对于一条用于训练的词序列长度为 l 语料数据，可以提供的 n 元词语组合的总数为 $(l-n+1)$ 。并且根据 Zipf 定律，少量词语占据了大部分的出现频次，若去除掉重复出现的 n 元组合，语料数据能提供的信息将更少。相对于语言模型的参数估计需求来说，语料数据是非常稀疏的。除非有海量的各种类型的语料数据，否则大量的 n 元组合都不曾在训练语料中出现过，

依据最大似然估计得到的概率将会是 0，也就是说模型可能仅仅能计算寥寥几个句子。对于这样的情况，学者们提出了多种不同的平滑方案进行估计，比如加一平滑（Add One Smoothing）、Good-Turing 平滑、Katz's back-off 平滑，以及插值模型等。

从实际应用方面来说，最常用的是二元 bigram（即 $n = 2$ ），或三元 trigram 即 $n = 3$ 。四元模型提高了时间复杂度，但在实际效果上来说并没有比三元模型有明显提高，所以一般不会使用。

5.5.2 神经网络语言模型

传统 NGram 语言模型存在比较大的问题。一方面，由于数据的稀疏性导致估算能力有限，仅能对两三个词的长度的序列进行评估；另一方面，由于 NGram 语言模型是离散模型，语义相似甚至一致的两个词语对于语言模型来说是完全不同的，例如“诺基亚”和“Nokia”就被视作两个不同的词语分别对待，虽然由此也引入一些针对性的改进版本，但并未从根本上解决类似问题。

为了解决这种问题，Yoshua Bengio 等人在 2003 年通过引入词向量概念，提出了基于神经网络的神经网络语言模型（NNLM, Neural Network Language Model）。

词向量的意思是使用一个向量来表示一个词。最直观也最常用的词向量就是 one-hot 编码，对词表中的每一个词进行编号，若词表中包含了 40000 常用词，则每一个单词都会编码为一个长度 40000 的向量，并且向量中只有一个元素的值为 1，其余全部为 0。one-hot 编码最主要的问题在于任意两个词之间都是孤立的，在语义上无法建立联系。另一种方式则是特征向量表示法。可以使用一个 2 层的全连接神经网络概述这一方法。字符串上下文构成训练样本数据集。（例如：单词 a,b,c 构成的语句字符串“abc a”，若以当前单词为目标类型，前后长度为 1 的单词作为特征可构成单词训练样本对“(b, a), (a, b), (c, b), (b, c), (a, c), (c, a)”）。对于，40000 个常用词的情况。收集大量的文本数据。每一对单词样本特征使用大小为 [1, 40000] 的 one-hot 编码特征输入 2 层全连接网络，网络的第一层是一个大小为 [40000, m] 的隐含层矩阵。为简化说明，这里忽略了线性偏移量。网络的的第二层是一个 [m, 40000] 的分类判定矩阵。因此，可直接使用单词样本上下文特征进行分类问题的模型训练。训练完毕后，可将第一层大小为 [40000, m] 的隐含层特征矩阵作为单词向量化的结果。因模型采用的是线性结构，可直接在单词向量化空间中使用空间距离度量单词的词义相同程度。与常

规的神经网络训练过程类似,整个神经网络的模型参数可使用方差较小的均匀分布或高斯分布进行初始化。

在神经网络语言模型中,单词的特征向量表示法也成为词向量。它是神经网络中对应模型参数,可以通过训练得到的。每一个词都会使用一个 m 维向量进行表示,并且语义相似的词语在特征空间上也会表示为相似的向量,在词空间中的距离较为相近。

词向量的相关方法有很多。在以上基于分类问题优化词向量的基础上,还可以使用更复杂的模型优化方法。例如,图 5-23 是 Bengio 等人 2001 年发表的论文 *A Neural Probabilistic Language Model* 中用于构造语言模型所使用的三层神经网络。该模型同样也是一个 NGram 模型,其中, $(w_{t-n+1}, \dots, w_{t-2}, w_{t-1})$ 代表前 $n-1$ 个词,要根据这些词来预测下一个词 w_t 。 $C(w)$ 表示词 w 所对应的词向量。在网络的输入层,词被转换成 $(n-1) \times m$ 大小的矩阵,接着就像正常的神经网络一样经过以 \tanh 为激活函数的非线性变换,最后通过 softmax 将输出值归一化成概率,就是对于下一个词 w_t 预测的条件概率 $p(w_t | \text{context})$ 。在这样的模型下,模型就可以像一般神经网络一样使用梯度下降算法进行优化,并通过训练得到最优的参数解。这其中,词向量的编码方式也是模型的参数,包含在 $C(w)$ 。也就是说,训练结束后,语言模型和词向量可以同时得到,如图 5-23 所示。

与传统的 NGram 语言模型比较,神经网络语言模型参数规模与词表规模 $|V|$ 和上下文依赖长度 n 成线性增长,在同等规模的语料库基础上,可以支持更长距离的上下文依赖。Bengio 在 APNews 数据集上做的对比实验也表明,神经网络语言模型的效果比精心设计平滑算法的普通 NGram 算法要好 10% ~ 20%。

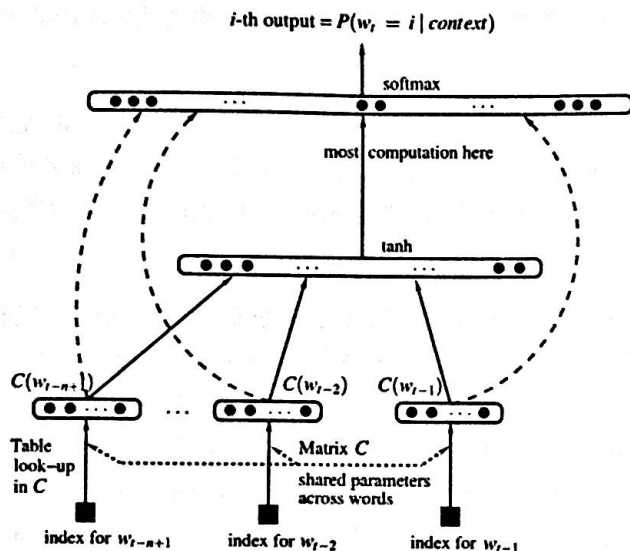


图 5-23 神经网络语言模型

5.5.3 循环神经网络语言模型

对于深度学习算法来说,要面临的一大核心问题就是如何减少参数个数,毕竟更少的参数可以有更快的收敛速度。语言模型也不例外, Bengio 在神经网络语言模型的论文中就有提出,可以利用循环神经网络来降低模型参数个数。2010 年,由 Tomas Mikolov 在 *Recurrent neural network based language model* 论文中提出了循环神经网络语言模型 (Recurrent Neural Network Language Model, RNNLM)。

在上一小节所讲的基于前馈神经网络的语言模型,仍然是以 NGram 模型为基础,要求在预测下一个词的时候依赖前 $n-1$ 个词的向量表示。从前文对循环神经网络的介绍中可以了解到, RNN 最主要的特点就是加入了“记忆”的因素,在预测当前时间点的结果时,会依赖之前时间所产生的记忆。在语言模型中, RNN 的这一特性也非常有用,通过引入 RNN,可以消除掉 n 个词的窗口限制,以全局的上下文信息预测当前词的概率。不仅如此,相比普通前馈神经网络来说, RNN 的参数共享可以大大减少模型的参数规模。

图 5-24 是循环神经网络模型的简化版示意图。可以看到,模型结构非常简单,

包含一个输入层，接收输入词的词向量；中间隐层保存上下文状态；输入与上下文相结合，产生输出为 one-hot 编码的词向量。

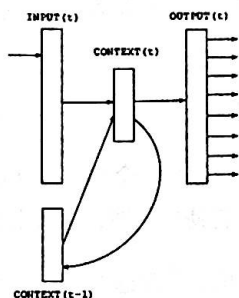


图 5-24 循环神经网络语言模型

图 5-25 是循环神经网络语言模型的展开形式。在图中，上文信息(w_1, w_2, \dots, w_{t-1})通过循环神经网络编码为 s_{t-1} ，是上一个隐藏层，代表着对上文的记忆。 s_{t-1} 与当前词语 w_t 相结合，可以得到(w_1, w_2, \dots, w_t)的表示 s_t ，通过编码得到预测的词向量 y_t 。

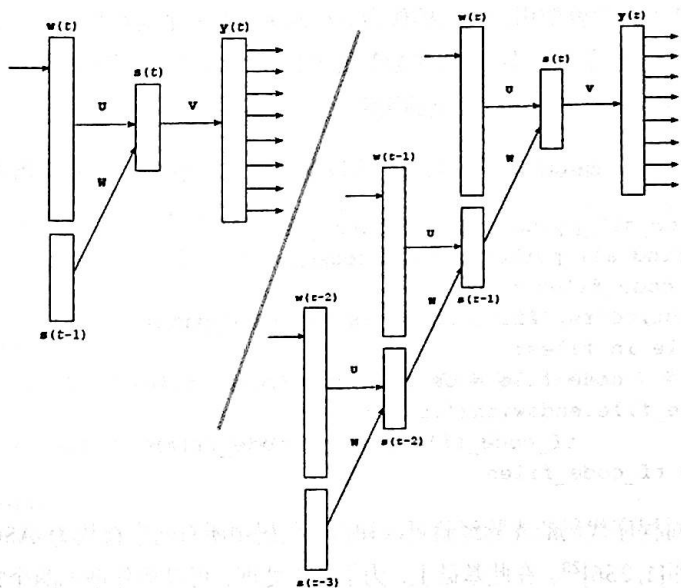


图 5-25 循环神经网络语言模型的展开形式

循环神经网络语言模型消除了 N-Gram 模型对于词窗口的限制，充分利用了完整

的上下文信息来进行预测评估,能达到比其他语言模型更好的效果。在 Tomas Mikolov 的论文中,即使使用最基础最简单的 RNN (原文为 *the simplest possible version of recurrent neural*) 和最基础的 BPTT 优化算法,也可以达到比 NGram 模型更好的效果,若使用更复杂的网络和更先进的优化算法,效果还会进一步提升。

5.5.4 语言模型也能写代码

在语言模型中,基础的预测单位是一个词也可以是一个字。利用语言模型,可以评估一段文本是否符合人类的语言使用习惯,也可以通过预测字符来生成一段程序。

什么?能写程序的程序?是的,就是这么科幻。下面就来介绍,基于 TensorFlow 的源码作为语料库,使用循环神经网络语言模型生成新代码的实例。

1. 训练数据准备

TensorFlow 是一个比较复杂而庞大的项目,整个系统由多种语言的程序构成。通过 Github,可以很方便地获取项目的源码。其中,Python 部分程序的文件总数达到了 1200+,在本例中就使用这些代码作为语料库来训练一个能写程序的语言模型。因篇幅限制,本节的后续代码将略去参数配置的相关代码。因这部分内容相对直观,本书配套的 github 源码中给出了完整的实例。

首先,在获取 TensorFlow 源码之后,可以从源码目录中遍历到所有 Python 代码:

```
def find_all_python_files(source_path):
    # find all python source code
    tf_code_files = []
    for root, dirs, files in os.walk(source_path):
        for file in files:
            code_file = os.path.join(root, file)
            if code_file.endswith('.py'):
                tf_code_files.append(code_file)
    return tf_code_files
```

以我们对程序代码的先验经验可以知道,代码中所有的字符均为 ASCII 码表的范围之内,即 $[1, 256]$ ²³。在此基础上,为了便于处理,可以额外增加两个特殊字符:

²³ ASCII 表中 0 为空字符,无法被直接打出。

BOF = 257 和 EOF = 258，分别代表代码文件起始和结束。

从字符粒度上看，每个代码文件可以表示为一个字符序列。按上述字符编码，程序文本实际上可以表示为一个整数序列。序列中的每个元素的取值都在 $[1, 258]$ 的范围之中，即词表（字符集）规模为 258^{24} 。利用下面代码片段，可以将所有 Python 文件读入成为字符序列，组成语料库，代码如下。

```
BOF = 257
EOF = 258

def read_source_code_data(code_files):
    data = []
    for code_file in code_files:
        file_r = open(code_file, 'r')
        curr_data = []
        curr_data.append(BOF)
        for dataline in file_r:
            for c in dataline:
                curr_data.append(ord(c))
        curr_data.append(EOF)
        data.extend(curr_data)
        file_r.close()
    return data
```

按照机器学习的一半方法，将整个语料数据划分成训练集 `train_data`、验证集 `valid_data` 和测试集 `test_data`。对于每一部分的数据集，将序列数据进行切割，并按 `batch_size` 组合成模型输入需要的 `tensor`，代码如下。

```
def tf_code_producer(raw_data, batch_size, num_steps, name=None):
    """
    This chunks up raw_data into batches of examples and returns
    Tensors
    that are drawn from these batches.

    Args:
        raw_data: one of the raw data outputs from tf_code_raw_data.
```

24 实际上，由于 ASCII 码表中的很多字符都不能被直接打出，代码文件中出现的字符集大小仅为 149，也就是所有能通过键盘打出的字符集合。利用这一条件可以进一步优化编码，缩小字符集的规模。考虑到简化实现，本书例子中仅把每个字符 `c` 直接映射为 `ord(c)`。

batch_size: int, the batch size.
 num_steps: int, the number of unrolls.
 name: the name of this operation (optional).

Returns:

A pair of Tensors, each shaped [batch_size, num_steps].
 The second element of the tuple is the same data time-shifted
 to the right by one.

```
"""
with tf.name_scope(name, "TensorFlowCodeProducer",
                    [raw_data, batch_size, num_steps]):
    raw_data = tf.convert_to_tensor(raw_data, name="raw_data",
                                    dtype=tf.int32)

    data_len = tf.size(raw_data)
    batch_len = data_len // batch_size
    data = tf.reshape(raw_data[0: batch_size * batch_len],
                      [batch_size, batch_len])

    epoch_size = (batch_len - 1) // num_steps
    assertion = tf.assert_positive(
        epoch_size,
        message="epoch_size == 0, decrease batch_size or
num_steps")
    with tf.control_dependencies([assertion]):
        epoch_size = tf.identity(epoch_size, name="epoch_size")

    i = tf.train.range_input_producer(epoch_size, shuffle=False).
    dequeue()
    x = tf.slice(data, [0, i * num_steps], [batch_size,
num_steps])
    x.set_shape([batch_size, num_steps])
    y = tf.slice(data, [0, i * num_steps + 1], [batch_size,
num_steps])
    y.set_shape([batch_size, num_steps])

    return x, y
```

以文件中的字符串为“Hello, TensorFlow!”为例，字符序列对应的整数数组为：

[257,72,101,108,108,111,44,32,84,101,110,115,111,114,70,108,111,119,33,258]

若设 batch_size 为 3，num_steps 为 4，则期望切割出大小为 3×4 的输入 tensor

和同样大小的输出 tensor。

首先根据 `batch_size = 3`, 把数列转换成 3×6 大小的矩阵。注意会忽略多余部分。

$$\begin{bmatrix} 257 & 72 & 101 & 108 & 108 & 111 \\ 44 & 32 & 84 & 101 & 110 & 115 \\ 111 & 114 & 70 & 108 & 111 & 119 \end{bmatrix}$$

其次根据 `num_steps = 4`, 模型输入大小 (此处指循环神经网络展开之后输入大小) 为 4 个一组。当输入为 `[257, 72, 101, 108]` 时, 其对应的输出应为右移一位的序列 `[72, 101, 108, 108]`。

`x = [[257, 72, 101, 108], [44, 32, 84, 101], [111, 114, 70, 108]]`

`y = [[72, 101, 108, 108], [32, 84, 101, 110], [114, 70, 108, 111]]`

2. 语言模型实现

TensorFlow 官方教程中有基于 Penn Tree Bank (简称 PTB) 数据集所构建的语言模型 `PTBModel`。在此例中, 可以借鉴其实现构建 `TFCodeModel`。`TFCodeModel` 类实现了模型的结构以及损失函数的定义。其中, `tf.nn.rnn_cell.BasicLSTMCell` 函数定义了一个隐含状态特征数和输出特征维数都为 `size` 的 LSTM 网络结构。LSTM 网络结构返回二元组, 元组中的两个分量分别都是长度为 `size` 的输出结果和状态特征。`tf.contrib.rnn.DropoutWrapper` 函数仅当训练时运行, 此时在网络结构中加入 dropout 层。`tf.nn.rnn_cell.MultiRNNCell` 是构建多个独立的串联循环神经网络结构, 其中每个循环神经网络使用独立的网络模型参数。该函数的输入参数 `[lstm_cell] × config.num_layers` 表示, 产生 `config.num_layers` 个独立的 LSTM 网络结构操作列表, 代码如下。

```
class TFCodeModel(object):
    """The TensorFlow python code language model."""
```

```
    def __init__(self, is_training, config, input_):
```

```
        self._input = input_
```

```
        batch_size = input_.batch_size
```

```
        num_steps = input_.num_steps
```

```
        size = config.hidden_size
```

```
        vocab_size = config.vocab_size
```

```
        # 语言模型使用多层以 LSTM 为基本单元的循环神经网络
```

```

lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units=size,
                                          state_is_tuple=True)
if is_training and config.keep_prob < 1:
    lstm_cell = tf.contrib.rnn.DropoutWrapper(
        lstm_cell, output_keep_prob=config.keep_prob)
cell = tf.nn.rnn_cell.MultiRNNCell(
    [lstm_cell] * config.num_layers, state_is_tuple=True)
# 初始隐含状态为 0
self._initial_state = cell.zero_state(batch_size, tf.float32)

```

在使用 LSTM 网络前使用 `cell.zero_state` 函数构建全零初始输出特征和状态特征。函数 `tf.nn.embedding_lookup` 负责从随机产生的初始词向量集合 `embedding` 中获取对应的样本特征，代码如下。

```

# 初始化词/字符向量表示（后续训练过程中会更新）
with tf.device("/cpu:0"):
    embedding = tf.get_variable(
        "embedding", [vocab_size, size], dtype=data_type())
    inputs = tf.nn.embedding_lookup(embedding,
                                    input_.input_data)
    if is_training and config.keep_prob < 1:
        inputs = tf.nn.dropout(inputs, config.keep_prob)

```

语句 `(cell_output, state) = cell(inputs[:, time_step, :], state)` 实现对 LSTM 网络的 1 次循环调用。因该语句在变量作用域“RNN”中调用，当在该作用域中使用 `for` 循环多次调用 LSTM 循环时必须配合使用 `tf.get_variable_scope().reuse_variables()` 函数确保网络节点参数的重复使用。否则，`cell(inputs[:, time_step, :], state)` 语句将产生同名网络节点导致报错，代码如下。

```

# 定义输出层
outputs = []
state = self._initial_state
with tf.variable_scope("RNN"):
    for time_step in range(num_steps):
        if time_step > 0: tf.get_variable_scope().reuse_
variables()
        (cell_output, state) = cell(inputs[:, time_step, :],
state)
        outputs.append(cell_output)
    output = tf.reshape(tf.concat(1, outputs), [-1, size])

```

`softmax_w` 和 `softmax_b` 是尺寸分别为 `[size, vocab_size]` 和 `[vocab_size]` 的随机张量。它们表示线性分类模型的参数。函数 `tf.nn.seq2seq.sequence_loss_by_example` 负责实现损失函数的计算。其中，输入参数 `[logits]` 表示预测的分类置信度结果。`[tf.reshape(input_.targets, [-1])]` 表示目标类型的 one-hot 编码张量。`[tf.ones([batch_size * num_steps], dtype=data_type())]` 表示各类型在损失函数的计算中权重全部相同都为 1，代码如下。

```
# 定义 softmax 层
softmax_w = tf.get_variable(
    "softmax_w", [size, vocab_size], dtype=data_type())
softmax_b = tf.get_variable("softmax_b", [vocab_size],
                             dtype=data_type())

# 定义损失函数
logits = tf.matmul(output, softmax_w) + softmax_b
loss = tf.nn.seq2seq.sequence_loss_by_example(
    [logits],
    [tf.reshape(input_.targets, [-1])],
    [tf.ones([batch_size * num_steps], dtype=data_type())])
self._cost = cost = tf.reduce_sum(loss) / batch_size
self._final_state = state

if not is_training:
    return
```

最后，函数 `tf.clip_by_global_norm` 实现梯度裁剪，防止梯度爆炸。`tf.train.GradientDescentOptimizer` 表示梯度下降算法。定义学习率占位符，在运行网络过程中可动态调整该参数。训练的文本样本信息由其他函数事先载入内存直接使用，代码如下。

```
# 定义优化模块，根据损失函数，计算梯度，使用梯度下降方法更新网络
self._lr = tf.Variable(0.0, trainable=False)
tvars = tf.trainable_variables()
grads, _ = tf.clip_by_global_norm(tf.gradients(cost, tvars),
                                   config.max_grad_norm)
optimizer = tf.train.GradientDescentOptimizer(self._lr)
self._train_op = optimizer.apply_gradients(
    zip(grads, tvars),
    global_step=tf.contrib.framework.get_or_create_
global_step())
```

```

self._new_lr = tf.placeholder(
    tf.float32, shape=[], name="new_learning_rate")
self._lr_update = tf.assign(self._lr, self._new_lr)

def assign_lr(self, session, lr_value):
    session.run(self._lr_update, feed_dict={self._new_lr:
lr_value})

```

语言模型采用评估指标：困惑度（Perplexity）。以输入文本 (w_1, w_2, \dots, w_n) 为例，语言模型LM的困惑度定义为

$$P = b^{-\frac{1}{n} \sum w_i \log_b p(w_i | \text{context})}$$

当对于所有 $p(w_i | \text{context})$ 均能精确预测正确时，也就是 $p(w_i | \text{context})$ 恒等于 1，这时 perplexity 最小为 0。当语言模型对所有 $p(w_i | \text{context})$ 进行随机预测时，这时 $p(w_i | \text{context})$ 均等于 $\frac{1}{|V|}$ （ $|V|$ 为词表规模，词表中任何一个词的可能性相同，均为 $\frac{1}{|V|}$ ）。这时困惑度为 $|V|$ 。可以看到，语言模型预测得越精确， $p(w_i | \text{context})$ 越大，困惑度越小。

3. 语言模型训练

语言模型的训练过程可以分成多轮，每轮训练遍历一遍所有训练数据。每轮训练过程包含 `model.input.epoch_size` 次模型训练，每次训练迭代的输入为 `batch_size * num_steps` 大小的 tensor。

训练过程通过反复调用 `run_epoch` 函数来完成。终止训练一般有几中控制方式：第一种是在到达固定训练迭代次数后终止，第二种是当验证集的评估代价指标达到某个阈值时终止，第三种方式是当验证集评估代价指标在连续几次迭代中的差距小于某个阈值，也就是训练不再有改进时终止。本例中采取第一种固定迭代轮数的方式，代码如下。

```

import time
import numpy as np

def run_epoch(session, model, eval_op=None, verbose=False):
    """Runs the model on the given data."""
    start_time = time.time()

```

```

costs = 0.0
iters = 0
state = session.run(model.initial_state)

fetches = {
    "cost": model.cost,
    "final_state": model.final_state,
}
if eval_op is not None:
    fetches["eval_op"] = eval_op

for step in range(model.input.epoch_size):
    feed_dict = {}
    for i, (c, h) in enumerate(model.initial_state):
        feed_dict[c] = state[i].c
        feed_dict[h] = state[i].h

    vals = session.run(fetches, feed_dict)
    cost = vals["cost"]
    state = vals["final_state"]

    costs += cost
    iters += model.input.num_steps

    if verbose and step % (model.input.epoch_size // 10) == 10:
        print("%.3f perplexity: %.3f speed: %.0f wps" %
              (step * 1.0 / model.input.epoch_size, np.exp(costs
/ iters),
              iters * model.input.batch_size /
              (time.time() - start_time)))
return np.exp(costs / iters)

```

类TFCodeInput实现训练数据的载入内存操作。相关代码定义如下。self.input_data, self.targets 分别代表单词序列特征和目标序列类型。

```

class TFCodeInput (object):
    """The input data."""
    def __init__(self, config, data, name=None):
        self.batch_size = batch_size = config.batch_size
        self.num_steps = num_steps = config.num_steps
        self.epoch_size = ((len(data) // batch_size) - 1) //
num_steps
        self.input_data, self.targets = tclm_reader.tensorflow_code_

```

```
producer(
    data, batch_size, num_steps, name=name)
```

函数 `tclm_reader.tensorflow_code_producer` 则负责实现数据的内存载入转换以及为模型提供 batch 训练数据。函数 `tf.convert_to_tensor` 实现 Python 数据与 TensorFlow 张量类型的转换。`epoch_size` 定义了循环 1 次完整数据集对应的 batch 训练数据包传入的次数。函数 `tf.assert_positive` 和 `tf.control_dependencies` 确保 `epoch_size` 必须为大于零的值，防止无效的参数输入。`tf.train.range_input_producer` 函数输入最大 batch 包的所在序号，返回一个独立线程负责运行线程整形队列出队列操作。最后 `tf.slice` 函数定义操作，从原始数据集中裁剪出 batch 数据包保存至类成员变量中，代码如下。

```
def tensorflow_code_producer(raw_data, batch_size, num_steps,
                             name=None):
    with tf.name_scope(name, "TensorflowCodeProducer", [raw_data,
                                                           batch_size, num_steps]):
        raw_data = tf.convert_to_tensor(raw_data, name="raw_data",
                                         dtype=tf.int32)

        data_len = tf.size(raw_data)
        batch_len = data_len // batch_size
        data = tf.reshape(raw_data[0:batch_size * batch_len], [batch_size,
                                                                batch_len])

        epoch_size = (batch_len - 1) // num_steps
        assertion = tf.assert_positive(
            epoch_size,
            message="epoch_size == 0, decrease batch_size or num_steps")
        with tf.control_dependencies([assertion]):
            epoch_size = tf.identity(epoch_size, name="epoch_size")

        i = tf.train.range_input_producer(epoch_size, shuffle=False).
        dequeue()
        x = tf.slice(data, [0, i * num_steps], [batch_size, num_steps])
        x.set_shape([batch_size, num_steps])
        y = tf.slice(data, [0, i * num_steps + 1], [batch_size,
                                                    num_steps])
        y.set_shape([batch_size, num_steps])
        return x, y
```

在 `main` 函数中使用了 3 次 `with tf.name_scope` 定义了训练、验证和测试操作。

`tf.random_uniform_initializer` 函数创建张量变量的随机初始化操作。函数 `tf.train.Supervisor` 是模型训练的集成操作集合类，它集成了 `tf.train.Saver` 和 `tf.Session` 等类操作。它的成员函数 `saver` 与 `tf.train.Saver` 类的对象对应。成员函数 `managed_session` 与 `tf.Session` 类的对象对应，代码如下。

```
def main():
    # 读入数据
    raw_data = read_source_code_data(FLAGS.source_path)
    train_data, valid_data, test_data = split_dataset(raw_data)

    config = get_config()
    eval_config = get_config()
    eval_config.batch_size = 1
    eval_config.num_steps = 1

    with tf.Graph().as_default():
        initializer = tf.random_uniform_initializer(-config.init_scale,
                                                    config.init_scale)

        with tf.name_scope("Train"):
            train_input = TFCodeInput(config=config, data=train_data,
                                       name="TrainInput")
            with tf.variable_scope("Model", reuse=None,
                                   initializer=initializer):
                m = TFCodeModel(is_training=True, config=config,
                               input_=train_input)
            tf.summary.scalar("Training Loss", m.cost)
            tf.summary.scalar("Learning Rate", m.lr)

        with tf.name_scope("Valid"):
            valid_input = TFCodeInput(config=config, data=valid_data,
                                       name="ValidInput")
            with tf.variable_scope("Model", reuse=True,
                                   initializer=initializer):
                mvalid = TFCodeModel(is_training=False, config=config,
                                     input_=valid_input)
            tf.summary.scalar("Validation Loss", mvalid.cost)

        with tf.name_scope("Test"):
            test_input = TFCodeInput(config=eval_config, data=test_
data,
                                   name="TestInput")
            with tf.variable_scope("Model", reuse=True,
```

```

initializer=initializer):
    mtest = TFCodeModel(is_training=False, config=eval_config,
                        input_=test_input)

    sv = tf.train.Supervisor(logdir=FLAGS.save_path)
    with sv.managed_session() as session:
        for i in range(config.max_max_epoch):
            lr_decay = config.lr_decay ** max(i + 1 -
            config.max_epoch, 0.0)
            m.assign_lr(session, config.learning_rate *
            lr_decay)

            print("Epoch: %d Learning rate: %.3f" % (
                i + 1, session.run(m.lr)))
            train_perplexity = run_epoch(session, m, eval_op=m.
            train_op,
            verbose=True)
            print("Epoch: %d Train Perplexity: %.3f" % (
                i + 1, train_perplexity))
            valid_perplexity = run_epoch(session, mvalid)
            print("Epoch: %d Valid Perplexity: %.3f" % (
                i + 1, valid_perplexity))

            test_perplexity = run_epoch(session, mtest)
            print("Test Perplexity: %.3f" % test_perplexity)

        if FLAGS.save_path:
            print("Saving model to %s." % FLAGS.save_path)
            sv.saver.save(session, FLAGS.save_path,
                          global_step=sv.global_step)

```

4. 言模型训练效果

上述模型训练过程中, 通过日志可以记录每轮模型训练之后, 训练集和验证集中的困惑度 perplexity 评估指标数值, 代码如下。

```

# Epoch: 1 Learning rate: 1.000
# 0.001 perplexity: 233.550 speed: 1458 wps
# 0.101 perplexity: 4.862 speed: 3676 wps
# 0.201 perplexity: 3.646 speed: 3879 wps
# 0.301 perplexity: 3.236 speed: 3950 wps
# 0.401 perplexity: 3.073 speed: 662 wps
# 0.501 perplexity: 3.010 speed: 161 wps

```



```
# 0.600 perplexity: 2.942 speed: 165 wps
# 0.700 perplexity: 2.891 speed: 155 wps
# 0.800 perplexity: 2.849 speed: 177 wps
# 0.900 perplexity: 2.813 speed: 198 wps
# Epoch: 1 Train Perplexity: 2.785
# Epoch: 1 Valid Perplexity: 4.057
# Epoch: 2 Learning rate: 1.000
.....
# Epoch: 10 Learning rate: 0.016
# Epoch: 10 Train Perplexity: 2.228
# Epoch: 10 Valid Perplexity: 2.721
# Epoch: 11 Train Perplexity: 2.232
# Epoch: 11 Valid Perplexity: 2.705
# Epoch: 12 Train Perplexity: 2.235
# Epoch: 12 Valid Perplexity: 2.695
# Epoch: 13 Train Perplexity: 2.235
# Epoch: 13 Valid Perplexity: 2.690
# Test Perplexity: 2.674
# Saving model to tc.lm.
```

可以看到随着训练轮数的增加，训练集和验证集中的困惑度指标都持续下降，并且在 10 轮训练之后已趋于平稳。

5.5.5 改进方向

对于语言模型，如果字符集规模比较小，简单的循环神经网络模型就可以得到比较不错的效果。但对于更普遍的应用场景来说，模型还可以从多种角度进行优化。

一方面，从词的表示来说，词表的选择可以是单词粒度或者是字符粒度。若以字符作为最小单元，则不需要考虑分词或单词归一化等问题²⁵，词表规模也比较小，但缺点是缺少了语义信息。而若以单词作为最小单元，则每个输入单元的语义信息得以保留，但词表规模会显著增大，且需要额外分词/Stemming 模块。词表规模是一个影响循环神经网络性能的重要因素，针对这一方面也有很多针对性的改进工作，如合并低频词为 rare token、词聚类、LightRNN 等技术。

除了单元表示粒度，词向量也是需要考虑的方面。对词的编码可以使用静态的映

²⁵ 分词问题如“Beijing Olympic”可以作为一个专有名词，也可以分成“Beijing”和“Olympic”两个词。单词归一化问题比如 book 和 books 应被归一化为同一个词。

射表，也可以是通过学习不断调整的动态编码。同时，one-hot 编码方式与稠密型编码方式也有效率上的差别。对于单词粒度的模型而言，因为词表规模相对比较大，若使用 one-hot 编码则会导致向量维度太高，所以一般情况下会采用 word embedding 技术的稠密向量表示。

另一方面，从模型结构来说，循环神经网络模型与其他深度学习模型一样，增加网络深度是必然的进化方向。但如同上一章介绍卷积神经网络类似，在增加网络深度的时候会遭遇梯度消失的问题，在实践中并不容易优化。另外，其它 RNN 和 LSTM 的变种模型，如双向循环神经网络 (Bidirectional RNNs) 等新模型，都是尝试的方向。

5.6 对话机器人

在当今的信息时代，计算机作为重要的信息传递工具已逐渐在全球范围普及。计算机在设计之初是用来执行一些人为设定的、规范的、无二义性的二进制指令。这些指令不过是简单的数字或逻辑运算。它们无法智能地理解并处理信息。但在这指令的基础上，通过一些算法控制，可以完成一些信息理解与处理的能力。例如，可以让计算机理解语言，并自动的返回对话结果。

这里所说的对话机器人 (Chatbot) 是指能够使用自然语言进行智能对话的软件系统。智能对话是自然语言处理领域的一个经典问题。类似的问题还有问答系统 (Query-Answer System)、会话系统 (Dialogue System) 等。虽然它们的内涵和外延不尽相同，但总体上，都是希望搭建支持自动对话的系统，让机器能以自然语言的方式与人类对话/沟通。

语言是人与人之间交流的主要方式。虽然科学家们尚没有完全理解人类是如何学习第一语言的整个过程。但学习过第二语言的人们都明白，语言学习分为多个阶段而且是一个漫长的过程。首先的初始阶段是达意，能够表达基本的需求或说明。比如，“吃饭”、“今天是礼拜天”。接着是在满足语法规则的前提下表达更复杂的意思。比如，“今天是礼拜天，我想出去吃饭”。最后，是在了解语言背景文化的基础上熟练掌握并运用语言。比如，“听说有家餐馆不错，总是要吃饭的，不如一起呗？”无论处于什么语言学习阶段。词汇量都是语言学习的关键基础。没有足够的词汇量，就无法熟练流畅的使用语言。人们在学习第二语言时，对于第二语言的词汇积累都是在第一语言

的转换基础上建立的。

然而, 计算机无法模拟人类学习语言的过程。计算机更擅长于科学计算以及各种数值表示和存储。很自然的, 人们想到使用数值信息让计算机自然语言的构建词汇信息。然后, 通过使用算法对数值进行操作来模拟词汇的运用。

5.6.1 对话机器人的发展

构建对话机器人的主要技术方案, 粗略分类, 可以划为三种: 基于规则, 基于检索和基于生成模型。

1. 基于规则的对话系统

基于规则对话机器人在实现算法上主要包括字符串匹配查找, 正则表达式等。AIML (Artificial Intelligence Markup Language) 是一种用于定义会话模板的 XML 语言, 历史上有多个基于 AIML 的对话机器人程序曾经获得罗布纳奖 (Loebner prize)²⁶, 比如 A.L.I.C.E (Artificial Linguistic Internet Computer Entity)。

下面是一个简单的 AIML 例子, 代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
  <aiml version="1.0">
    <meta name="author" content="Mobo"/>
    <meta name="language" content="zh"/>
    <category>
      <pattern>拜拜</pattern>
      <template>下次见~~~</template>
    </category>
    <category>
      <pattern>谢谢</pattern>
      <template>
        <srai>THANKS</srai>
      </template>
    </category>
    <category>
      <pattern>谢谢 *</pattern>
      <template>
```

²⁶ 罗布纳奖是纽约慈善家 Hugh Loebner 从 1991 年开始组织的正式图灵测试。

```

<srai>THANKS</srai>
</template>
</category>
<category>
<pattern>THANKS</pattern>
<template>
<random>
<li>不用谢。</li>
<li>这是我应该做的。</li>
<li>客气，客气~</li>
</random>
</template>
</category>
</aiml>
</>

```

上述 AIML 中指定了一系列对话的规则。比如当输入为“拜拜”时，程序回复“下次见~~”；当输入为“谢谢”或者符合正则表达式“谢谢*”时，随机回复“不用谢”、“这是我应该做的。”、“客气，客气~”中的一条。

基于规则的算法，受限于规则的数量，还受限于正则表达式的表达能力和匹配效率。不同于数学推导，自然语言在不同的语境下表达的是完全不同的意思。比如，“我挂了”在打电话过程中表示是结束通话的意思，而在学生们讨论结果时则表示考试不及格。若想在 AIML 模板中定义文字短语在不同语境回复不同的应答。则需要编写复杂的正则表达式规则模板，枚举出每一个语境对应的关键词进行匹配关联。若询问语句不存在对应的匹配规则，则无法获得结果。在实践中常使用语料数据挖掘算法自动构建 AIML 配置数据，来减少人工标注的工作量。但不管这样，这种应答方式始终都是根据人工经验设置的。算法只负责人工经验与机器的信息转换，机器并没有学习获取语言使用的能力。

2. 基于检索的对话系统

定义复杂的会话规则是十分烦琐且困难的，对此基于检索的对话系统则进行了改进。2014年华为的研发工程师曾发表了论文提出了一种3级结构的短消息对话系统²⁷。

27 An Information Retrieval Approach to Short Text Conversation, Zongcheng Jia, Zhengdong Lu, Hang Li, 参见本章参考资料[8]。

该系统从社交媒体网站如 (weibo.com 和 twitter.com 等网站上) 上获取并构建庞大的短句子 (weibo.com 用户仅能发表或回复长度小于 140 个字的短消息) 应答对话信息集合 (p, r) 。其中, p 表示提问语句, r 表示对应的回答语句。基于检索的对话系统将从庞大的对话信息中挑选出最合适回答语句返回给提问者。

系统结构如图 5-26 所示, 共由三大模块组成, 步骤如下:

步骤 1 首先使用归一化词频向量, 将提问短消息 $q(\text{query})$ 与数据集 (post-comment pairs) 中的提问语句 p 、回答语句 r 的分别进行余弦匹配分别获取 10 个相似度最高的对话信息样本。另外, 短消息 q 和回答语句 r 再进行一次增强线性匹配获取 10 个相似度最高的对话信息样本。获取 30 个候选样本 (p_i, r_i) 。

步骤 2 利用语言模型、关键词匹配模型预测等复杂方法获取每一个候选样本 (p_i, r_i) 与提问语句 q 的匹配 (matching) 关联特征。因对话信息集合 (p, r) 中存在多个回复消息 r 对应同一个原始提问语句 p 的情况。因此, 在步骤 1 中使用原始提取语句检索即可针对性的收集正、负样本信息。语言模型、关键词匹配模型可据此进行数据训练。

步骤 3 利用步骤 2 中计算获得匹配特征, 使用线性函数计算提问语句 q 与候选样本 (p_i, r_i) 间的相似度值并进行排序 (ranking)。最后, 返回相似度最高的值。排序算法使用的是类型数量为 2 数据进行训练的。训练数据的类型设置与步骤 2 相同。

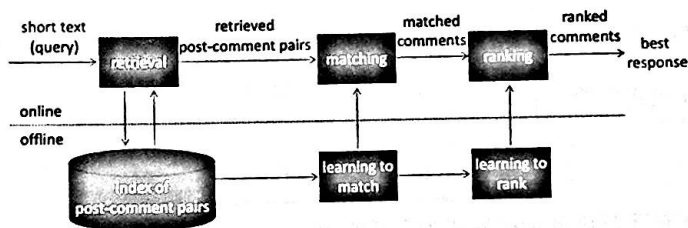


图 5-26 基于检索的对话机器人系统结构

该方法所呈现的处理结果将呈现出更加智能的效果。与规则系统相比较, 检索系统的回答方式完全有庞大的短消息数据集驱动, 极大地减少了规则系统中规则调整的工作量。但这种方式, 从本质上说解决的依然是一个语言检索问题。算法并没有真正学习并利用语言的内在逻辑关系。在数据集和相关参数保持不变的情况下, 反复输入相同的语句将返回完全一样的相同结果。而且该系统结构复杂, 样本维护种类较多和

模型的训练较为分散，这些令最终的系统调试与评估十分繁琐。

3. 基于生成模型

前面介绍的循环神经网络 RNN, LSTM 和 GRU, 为我们处理不定长时序数据提供了基础工具, 可以很方便地应用到 5.2.3 节中的 one to many, many to one 和 many to many(sync)应用场景。对于异步形式的 many to many, 由于输入和输出数据规模并不一致, 不能使用单一循环神经网络解决。Seq2seq 方法是一种基于 Encoder-Decoder 框架和循环神经网络的面向 many to many 异步形式场景的解决方案。

Encoder-Decoder 框架 (如图 5-27 所示) 包括两部分, 第一部分将输入信息通过 Encoder 模块编码为中间表示特征 w ; 第二部分使用 Decoder 模块将中间特征 C 解码获取预测结果。在编码和解码预测的过程就是自循环神经网络的使用过程。特别的, 在解码的开始时会输入开始预测标记特征 $\langle go \rangle$, 当预测结果返回结束标记 $\langle eos \rangle$ 或超过语句最大值时, 预测结果结束。

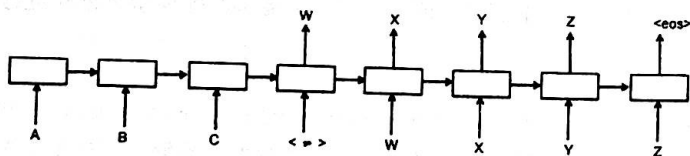


图 5-27 Encoder-Decoder 框架

图 5-27 中输入 $I = (A, B, C)$, 输出 $O = (W, X, Y, Z)$ 。Encoder-Decoder 框架中, Encoder 负责将输入转化为内部编码:

$$\text{IntermediateCode} = \text{Encode}(I)$$

Decoder 负责将内部编码解码为合适的输出:

$$O = \text{Decode}(\text{IntermediateCode}) = \text{Decode}(\text{Encode}(I))$$

由此可见, Seq2seq 方法非常适用于智能对话问题。通过收集对话样本, 将提问语句编码为中间特征, 令回复结果为类型序列标签, 进行循环网络的模型训练即可直接实现智能对话模型。标准的 RNN 模型能够通过隐含层的状态信息提取样本序列的相关特征。而 LSTM 模型具有自适应记忆、遗忘功能, 在样本序列的训练过程中能

够更好地获取特征信息。

不管是基于规则，还是基于检索的方案，都将只返回预先定义的回答内容。而基于生成模型的算法，可以根据模型训练获取的语义关联信息生成全新的未曾见过的回复。

5.6.2 基于 seq2seq 的对话机器人

下面重点介绍如何基于 Sequence to sequence 模型搭建基于生成模型的对话机器人。

1. 训练数据

对于端对端训练神经会话模型，我们需要类似机器翻译平行语料的问答对。一般构建对话系统之初，我们需要从网络中收集类似的训练语料。比如：电影对白，问答网站，社交网站数据等。

这里以华为诺亚方舟实验室公开的微博数据为例（如图 5-28 所示），微博数据包括用户所发出的微博（Posts）和该帖子下方的若干用户评论（Comments）。组合每个帖子和评论对，以 Post-Comment 对，作为问答对（问题-回复）数据。

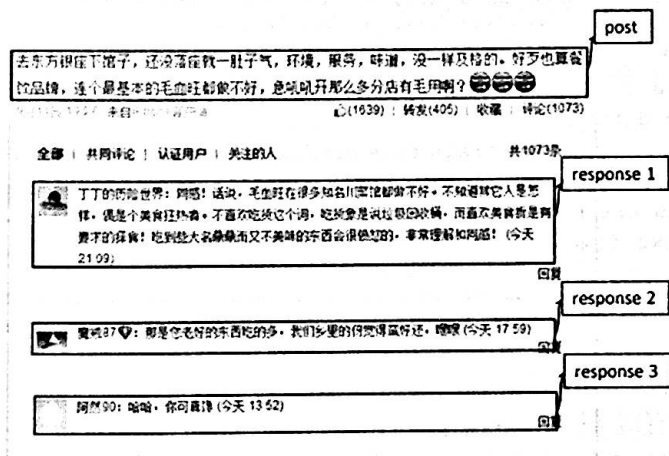


图 5-28 微博数据

将问题数据存储为 enquiry 文件，可以看到其中一些经过分词的例子：

```
$ head -10 enquiry
```

小马也疯狂 ----- 地位之争。

那些年，我们一起偷看过的电视。「暴走漫画」

北京的小纯洁们，周日见。 # 硬汉摆拍清纯照 #

要是这一年哭泣的理由不再是难过而是感动会多么好

对于国内动漫画作者引用工笔素材的一些个人意见。

同样，可以看到部分回复数据的例子：

```
$ head -10 answer
```

王大姐，打字细心一点

于老师不给劝劝架么告诉他们再挣也不是老大

真不愧是这么走出来的少年.....

嗷嗷大神的左脚在干什么，看着小纯洁带球么。

我已经快感动得哭了。

问题和回复数据文件逐行匹配，得到问答对。比如：

Q：如果有个人能让你忘掉过去，那 ta 很可能就是你的未来。

A：关键是那人是否忘记他的过去。

对原始对话数据，进行中文分词，转化成词 ID，并分拆为训练集和验证集，代码如下：

```
def prepare_data(data_dir, vocabulary_size, train_file='train',
                 dev_file='dev', use_fmm_tokenizer=False):
    """
    prepare dialog data. all the data should be put in the data_dir
    :param data_dir:
    :param train_file: train file prefix. there should be two train-file
    prefix files. By default, you should name the enquiry
    file as train.enquiry, the answer file as train.answer. each line
    of these two file make a enquiry, answer pair.
    :param dev_file: almost the same as train file, except that it is
    used to internal evaluate
    :param vocabulary_size:
    :return:
    """
    tokenizer = fmm_tokenizer if use_fmm_tokenizer else basic_tokenizer
    # Create vocabularies of the appropriate sizes.
    vocab_path = os.path.join(data_dir, "vocab%d" % vocabulary_
size)
    create_vocabulary(vocab_path, data_dir + '/test', vocabulary_
```



```

size,
tokenizer=ttokenizer)

# Create token ids for the training data.
enquiry_train_ids_path = os.path.join(data_dir, train_file + (
    ".ids%d.enquiry" % vocabulary_size))
answer_train_ids_path = os.path.join(data_dir, train_file + (
    ".ids%d.answer" % vocabulary_size))
data_to_token_ids(os.path.join(data_dir, train_file +
    ".enquiry"),
                  enquiry_train_ids_path, vocab_path, ttokenizer)
data_to_token_ids(os.path.join(data_dir, train_file + ".answer"),
                  answer_train_ids_path, vocab_path, ttokenizer)

# Create token ids for the development data.
enquiry_dev_ids_path = os.path.join(data_dir, dev_file + (
    ".ids%d.enquiry" % vocabulary_size))
answer_dev_ids_path = os.path.join(data_dir, dev_file + (
    ".ids%d.answer" % vocabulary_size))
data_to_token_ids(os.path.join(data_dir, dev_file +
    ".enquiry"),
                  enquiry_dev_ids_path, vocab_path, ttokenizer)
data_to_token_ids(os.path.join(data_dir, dev_file +
    ".answer"),
                  answer_dev_ids_path, vocab_path, ttokenizer)

return (enquiry_train_ids_path, answer_train_ids_path,
        enquiry_dev_ids_path, answer_dev_ids_path,
        vocab_path)

```

其中，主要逻辑 `data_to_token_ids` 实现从文本到词 ID 系列的转换过程，代码如下：

```

def sentence_to_token_ids(sentence, vocabulary,
tokenizer, normalize_digits=True):
    """Convert a string to list of integers representing token-ids.

    For example, a sentence "I have a dog" may become tokenized into
    ["I", "have", "a", "dog"] and with vocabulary {"I": 1, "have":
2,
    "a": 4, "dog": 7"} this function will return [1, 2, 4, 7].

Args:

```

sentence: a string, the sentence to convert to token-ids.
 vocabulary: a dictionary mapping tokens to integers.
 tokenizer: a function to use to tokenize each sentence;
 if None, basic_tokenizer will be used.
 normalize_digits: Boolean; if true, all digits are replaced
 by 0s.

Returns:

a list of integers, the token-ids for the sentence.

```
"""
words = tokenizer(sentence) if tokenizer else basic_tokenizer
(sentence)
if not normalize_digits:
    return [vocabulary.get(w, UNK_ID) for w in words]
    # Normalize digits by 0 before looking words up in the vocabulary.
return [vocabulary.get(re.sub(_DIGIT_RE, "0", w), UNK_ID) for w in
words]
```

```
def data_to_token_ids(data_path, target_path, vocabulary_path,
tokenizer, normalize_digits=True):
    """Tokenize data file and turn into token-ids using given
    vocabulary file.
```

This function loads data line-by-line from data_path, calls the
 above

sentence_to_token_ids, and saves the result to target_path. See
 comment

for sentence_to_token_ids on the details of token-ids format.

Args:

data_path: path to the data file in one-sentence-per-line
 format.

target_path: path where the file with token-ids will be created.

vocabulary_path: path to the vocabulary file.

tokenizer: a function to use to tokenize each sentence;

if None, basic_tokenizer will be used.

normalize_digits: Boolean; if true, all digits are replaced
 by 0s.

"""

```
if gfile.Exists(target_path):
```

```
sys.stderr.write(
```

```
'target path %s already exist! we will use the existed
```

```

one.\n' % target_path)
else:
    print("Tokenizing data in %s" % data_path)
    vocab, _ = initialize_vocabulary(vocabulary_path)
    with gfile.GFile(data_path, mode="r") as data_file:
    with gfile.GFile(target_path, mode="w") as tokens_file:
        counter = 0
        for line in data_file:
            counter += 1
            if counter % 100000 == 0:
                print(" tokenizing line %d" % counter)
                token_ids = sentence_to_token_ids(line, vocab,
tokenizer,
normalize_digits)
                tokens_file.write(
                    " ".join([str(tok) for tok in token_ids]) + "\n")

```

2. 会话模型实现

会话模型基于 TensorFlow 中的 Seq2SeqModel。利用下面的 create_model 方法中可以直接加载事先训练好的会话模型，代码如下。

```

def create_model(session, forward_only):
    """Create conversation model and initialize or load parameters
in session."""
    model = seq2seq_model.Seq2SeqModel(
        FLAGS.vocab_size, FLAGS.vocab_size, _buckets,
        FLAGS.size, FLAGS.num_layers, FLAGS.max_gradient_norm,
FLAGS.batch_size,
        FLAGS.learning_rate, FLAGS.learning_rate_decay_factor,
        use_lstm=FLAGS.use_lstm,
        forward_only=forward_only)
    ckpt = tf.train.get_checkpoint_state(FLAGS.train_dir)
    if ckpt and gfile.Exists(ckpt.model_checkpoint_path):
        print("Reading model parameters from %s" % ckpt.model_checkpoint_
path)
        model.saver.restore(session, ckpt.model_checkpoint_path)
    else:
        print("Created model with fresh parameters.")
        session.run(tf.initialize_all_variables())
    return model

```

3. 会话模型训练

会话模型的训练过程并不复杂，加载训练语料之后，直接调用 Seq2SeqModel 训练流程，代码如下。

```
def train():
    # Prepare conversation data.
    print("Preparing conversation data in %s" % FLAGS.data_dir)
    enquiry_train, answer_train, enquiry_dev, answer_dev, _ =
data_utils.prepare_data(
    FLAGS.data_dir, FLAGS.vocab_size)
    vocab_path = os.path.join(FLAGS.data_dir, "vocab%d" %
FLAGS.vocab_size)
    vocab, rev_vocab = data_utils.initialize_vocabulary(vocab_path)

    with tf.Session() as sess:
        # Create model.
        print(
            "Creating %d layers of %d units." % (FLAGS.num_layers,
FLAGS.size))
        model = create_model(sess, False)
        # Read data into buckets and compute their sizes.
        print("Reading development and training data (limit: %d)."
            % FLAGS.max_train_data_size)
        dev_set = read_data(enquiry_dev, answer_dev)
        train_set = read_data(enquiry_train, answer_train,
FLAGS.max_train_data_size)

        train_bucket_sizes = [len(train_set[b]) for b in
xrange(len(_buckets))]
        train_total_size = float(sum(train_bucket_sizes))
        # A bucket scale is a list of increasing numbers from 0 to
1 that we'll use
        # to select a bucket. Length of [scale[i], scale[i+1]] is
proportional to
        # the size if i-th training bucket, as used later.
        train_buckets_scale = [sum(train_bucket_sizes[:i + 1]) /
train_total_size
            for i in xrange(len(train_bucket_sizes))]

        # This is the training loop.
        print("Start training ...")
        step_time, loss = 0.0, 0.0
        current_step = 0
```

```

previous_losses = []
while True:
    # Choose a bucket according to data distribution. We pick
    a random number
    # in [0, 1] and use the corresponding interval in
    train_buckets_scale.
    random_number_01 = np.random.random_sample()
    bucket_id = min([i for i in
xrange(len(train_buckets_scale))
        if train_buckets_scale[i] >
random_number_01])
    # Get a batch and make a step.
    start_time = time.time()
    encoder_inputs, decoder_inputs, target_weights =
model.get_batch(
        train_set, bucket_id)
    _, step_loss, _ = model.step(sess, encoder_inputs,
decoder_inputs,
        target_weights, bucket_id,
False)
    step_time += (time.time() - start_time) / FLAGS.steps_
per_checkpoint
    loss += step_loss / FLAGS.steps_per_checkpoint
    current_step += 1
    # Once in a while, we save checkpoint, print statistics,
    and run evals.
    if current_step % FLAGS.steps_per_checkpoint == 0:
        # Log
        log_head = 'current_step: %s' % model.global_step.
eval()
        # Print statistics for the previous epoch.
        perplexity = math.exp(loss) if loss < 300 else float('inf')
        print(
            "global step %d learning rate %.4f step-time %.2f
perplexity "
            "%.2f" % (
                model.global_step.eval(), model.learning_rate.eval(),
                step_time, perplexity))
        # Decrease learning rate if no improvement was seen
        over last 3 times.
        if len(previous_losses) > 2 and loss > max(
            previous_losses[-3:]):
            sess.run(model.learning_rate_decay_op)

```

```

previous_losses.append(loss)
# Save checkpoint and zero timer and loss.
checkpoint_path = os.path.join(FLAGS.train_dir,
                                "conversation.ckpt")
model.saver.save(sess, checkpoint_path,
                  global_step=model.global_step)
step_time, loss = 0.0, 0.0
# Run evals on development set and print their
perplexity.
for bucket_id in xrange(len(_buckets)):
    encoder_inputs, decoder_inputs, target_weights =
model.get_batch(
    dev_set, bucket_id)
    _, eval_loss, _ = model.step(sess, encoder_inputs,
                                decoder_inputs,
                                target_weights,
                                True)
    eval_ppx = math.exp(
        eval_loss) if eval_loss < 300 else float('inf')
    print(" eval: bucket %d perplexity %.2f" % (
        bucket_id, eval_ppx))
    # Log the answer of validation set: use 20 enquiries
from development set
    for bucket_group in dev_set:
        # decode 4 questions in each bucket
        for pair in bucket_group[:4]:
            token_ids = pair[0]
            log_info = '%s, enquiry: %s' % (log_head, "".join(
                [rev_vocab[inp] for inp in token_ids]))
            # Which bucket does it belong to?
            bucket_id = min([b for b in xrange(len(_buckets))
                            if _buckets[b][0] >
len(token_ids)])
            # Get a 1-element batch to feed the sentence
to the model.
            encoder_inputs, decoder_inputs, target_weights
= model.get_batch(
                {bucket_id: [(token_ids, [])]}, bucket_id)
            # Get output logits for the sentence.
            _, _, output_logits = model.step(sess, encoder_
inputs,
                                decoder_inputs,

```

```

        target_weights,
        bucket_id, True)
    # This is a greedy decoder - outputs are just
    argmaxes of output_logits.
    # Batch_size = 64, we select the first output_
    logit
    outputs = [int(np.argmax(logit, axis=1)[0]) for logit in
                output_logits]
    # If there is an EOS symbol in outputs, cut them
    at that point.
    # if data_utils.EOS_ID in outputs:
    #                                     outputs =
    outputs[:outputs.index(data_utils.EOS_ID)]
    log_info = '%s, answer: %s' % (log_info,
    "".join([rev_vocab[output] for output in outputs]))
    print(log_info)
    # _LOGGER.info(log_info)
    sys.stdout.flush()

```

需要注意的是，加载训练数据部分，从训练效率角度考虑，把不同长度问答数据进行分桶处理，代码如下：

```

# We use a number of buckets and pad to the closest one for efficiency.
# See seq2seq_model.Seq2SeqModel for details of how they work.
_buckets = [(5, 10), (10, 15), (20, 25), (40, 50)]

def read_data(source_path, target_path, max_size=None):
    """Read data from source and target files and put into buckets.
    Args:
        source_path: path to the files with token-ids for the source
        language.
        target_path: path to the file with token-ids for the target
        language;
        it must be aligned with the source file: n-th line contains the
        desired
        output for n-th line from the source_path.
        max_size: maximum number of lines to read, all other will be
        ignored;
        if 0 or None, data files will be read completely (no limit).
    Returns:
        data_set: a list of length len(_buckets); data_set[n] contains
        a list of

```

```

        (source, target) pairs read from the provided data files that
fit
    into the n-th bucket, i.e., such that len(source) < _buckets[n][0]
and
    len(target) < _buckets[n][1]; source and target are lists of
token-ids.
    """
    data_set = [[] for _ in _buckets]
    with gfile.GFile(source_path, mode="r") as source_file:
    with gfile.GFile(target_path, mode="r") as target_file:
        source, target = source_file.readline(), target_file.readline()
        counter = 0
        while source and target and (not max_size or counter < max_size):
            counter += 1
            if counter % 1000 == 0:
                print(" reading data line %d" % counter)
                sys.stdout.flush()
                source_ids = [int(x) for x in source.split()]
                target_ids = [int(x) for x in target.split()]
                target_ids.append(tokenizer.EOS_ID)
            for bucket_id, (source_size, target_size) in enumerate(
                _buckets):
                if len(source_ids) < source_size and len(
                    target_ids) < target_size:
                    data_set[bucket_id].append([source_ids,
target_ids])
            break
        source, target = source_file.readline(), target_file.readline()
    return data_set

```

4. 会话模型训练效果

基于训练好的会话模型，调用 Seq2SeqModel 的 decode 过程，可以对任意新输入的问题，自动生成新的回复，代码如下：

```

class Chatbot():
    """
    answer an enquiry using trained seq2seq model
    """

    def __init__(self, model_dir):
        # Create model and load parameters.
        self.session = tf.InteractiveSession()

```



```

self.model = self.create_model(self.session, model_dir,
True)

self.model.batch_size = 1
# Load vocabularies.
vocab_path = os.path.join(FLAGS.data_dir, "vocab%d" %
FLAGS.vocab_size)
self.vocab, self.rev_vocab =
data_utils.initialize_vocabulary(vocab_path)

def create_model(self, session, model_dir, forward_only):
    """Create conversation model and initialize or load
parameters in session."""
    model = seq2seq_model.Seq2SeqModel(
        FLAGS.vocab_size, FLAGS.vocab_size, _buckets,
        FLAGS.size, FLAGS.num_layers, FLAGS.max_gradient_norm,
        FLAGS.batch_size,
        FLAGS.learning_rate, FLAGS.learning_rate_decay_factor,
        use_lstm=FLAGS.use_lstm,
        forward_only=forward_only)

    ckpt = tf.train.get_checkpoint_state(model_dir)
    if ckpt and tf.train.checkpoint_exists(ckpt.model_checkpoint_path):
        _LOGGER.info("Reading model parameters from %s" %
ckpt.model_checkpoint_path)
        model.saver.restore(session, ckpt.model_checkpoint_path)
        _LOGGER.info("Read model parameter succeed!")
    else:
        raise ValueError(
            "Failed to find legal model checkpoint files in %s" %
model_dir)
    return model

def generate_answer(self, enquiry):
    # Get token-ids for the input sentence.
    token_ids = data_utils.sentence_to_token_ids(enquiry,
self.vocab, tokenizer.fmm_tokenizer)
    if len(token_ids) == 0:
        _LOGGER.error('lens of token ids of sentence %s is 0' %
enquiry)

    # Which bucket does it belong to?
    bucket_id = min([b for b in xrange(len(_buckets))
        if _buckets[b][0] > len(token_ids)])
    # Get a 1-element batch to feed the sentence to the model.

```

```

        encoder_inputs, decoder_inputs, target_weights = self.model.get_batch(
            (bucket_id: [(token_ids, [])]), bucket_id)
        # Get output logits for the sentence.
        _, _, output_logits = self.model.step(self.session,
            encoder_inputs,
            decoder_inputs,
            target_weights, bucket_id,
            True)
        # This is a greedy decoder - outputs are just argmaxes of
        output_logits.
        outputs = [int(np.argmax(logit, axis=1)) for logit in
            output_logits]
        # If there is an EOS symbol in outputs, cut them at that point.
        if tokenizer.EOS_ID in outputs:
            outputs = outputs[:outputs.index(tokenizer.EOS_ID)]
        # Print out response sentence corresponding to outputs.
        answer = " ".join([self.rev_vocab[output] for output in outputs])
        if tokenizer._UNK in answer:
            answer = random.choice(UNK_TOKEN_REPLACEMENT)
        answer = trim(answer)
        return answer

    def close(self):
        self.session.close()

if __name__ == "__main__":
    m = Chatbot(SEQ2SEQ_MODEL_DIR + '/train/')
    response = m.generate_answer(u'我知道你不知道我知道你不知道我说的是
        什么意思')
    print response

```

效果不错的自动回复例子:

Q: 北鼻, 情人节快乐! 来, 嘴一个。

A(Bot): 好可爱的小盆友。

Q: 《财经郎眼》上周蝉联优酷播放榜第一名

A(Bot): 我也在看, 很期待!

Q: 还记得你看的第一场演唱会是谁的吗?

A(Bot): 当然是周杰伦!

然而,也有一些质量不理想的回复例子:

Q: 有些人,表面装的对你很好,实际呢?「转」

A(Bot): 有些人,有些人,有些人,有些人,有些人。

Q: 知识就是力量。(这得看到何年何月啊)

A(Bot): 这就是传说中的“”字“”的意思

Q: 艾玛,少女心的颜色 ...

A(Bot): 我也是,我是颜色颜色

5.7 小结

循环神经网络 RNN 是目前在处理序列数据的优秀模型,在处理自然语言处理领域的文本理解、文本生成等问题上的应用已十分广泛。理论上,循环神经网络已经被证明是图灵完备(Turing-Complete)的,RNN 有潜力可以模拟任何程序。虽然现实和理论仍然存在巨大差距,但在实践中,RNN 在不同问题上的应用还是展现了非常多出人意料的效果,并且 LSTM 等技术还在持续改进 RNN 的效果。

从另一个维度,基于循环神经网络的 sequence to sequence 模型在神经会话模型、机器翻译等领域都相继取得了突破性的成果。尤其在机器翻译领域,从最初的基于规则的机器翻译系统,到基于统计的机器翻译系统(SMT, Statistical Machine Translation),再到神经网络机器翻译(NMT, Neural machine translation),直到 2016 年 Google 推出的集大成的 GNMT (Google's Neural Machine Translation System),机器翻译的质量有了大幅度的提高。从结构上讲,GNMT 仍然遵循 sequence to sequence 框架,只不过在此基础上融入了诸如注意力模型、残差连接、多层 LSTM 网络、迁移学习等一系列技术,才成就了今天 Google Translate 这一逆天的产品。

相信在更多应用场景中,RNN 和 Sequence to sequence 模型还将继续给我们带来更多惊喜。

5.8 参考资料

- [1] Ian Goodfellow and Yoshua Bengio and Aaron Courville. "Deep learning." An MIT Press book in preparation. Draft chapters available at <http://deeplearningbook.org/> (2016).
- [2] <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- [3] <http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-back-propagation-through-time-and-vanishing-gradients/>
- [4] <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>
- [5] Bengio, Yoshua, et al. "A neural probabilistic language model." Journal of machine learning research 3.Feb (2003): 1137-1155.
- [6] Mikolov, Tomas, et al. "Recurrent neural network based language model." Interspeech. Vol. 2. 2010.
- [7] Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks." ICML (3) 28 (2013): 1310-1318.
- [8] Ji, Zongcheng, Zhengdong Lu, and Hang Li. "An information retrieval approach to short text conversation." arXiv preprint arXiv:1408.6988 (2014).
- [9] DocChat: An Information Retrieval Approach for Chatbot Engines Using Unstructured Documents
- [10] Shang, Lifeng, Zhengdong Lu, and Hang Li. "Neural responding machine for short-text conversation." arXiv preprint arXiv:1503.02364 (2015).
- [11] Vinyals, Oriol, and Quoc Le. "A neural conversational model." arXiv preprint arXiv:1506.05869 (2015).
- [12] Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in NIPS, 2014.

6

CNN+LSTM 看图说话

作为人工神经网络发展的里程碑，CNN 与 LSTM 分别是在计算机图像识别与自然语言处理这两个相对独立的领域诞生的。但这并不意味着 CNN 与 LSTM 的应用没有丝毫联系。CNN 的特点是通过卷积操作共享网络层参数，在减少网络参数数量的同时，能够有效提取图像局部、甚至是整体的特征信息。LSTM 是一种拥有长、短期自适应记忆、遗忘能力的循环神经网络。与非循环神经网络的最大区别在于网络输入、输出接口上的差异。一般非循环神经网络结构（如 CNN 或全连接网络）只能提取出单个样本信息的内在关联信息，输出结果仅与当前输入的样本信息相关，而 LSTM 则可以训练获取存在于样本序列之间的关联信息，若将同一条样本输入同一个 LSTM 网络两次，输出结果可能是不同的。

尽管 CNN 和 LSTM 早先被应用于不同的领域，但这两者的结合使用已在图像检测和图像摘要问题中得到了成功的应用。图像检测问题已经在第 4 章有过介绍，主要解决的是定位图像中所有目标物体。图 6-1 左图显示的是图像检测问题中的人脸检测结果示意图。图像摘要问题是指对于给定的图片，计算机能识别其中的内容，并且根据内容的语义生成一段能够描述图片的文字。图 6-1 右图显示了图像摘要算法自动生成描述图像语句的例子。

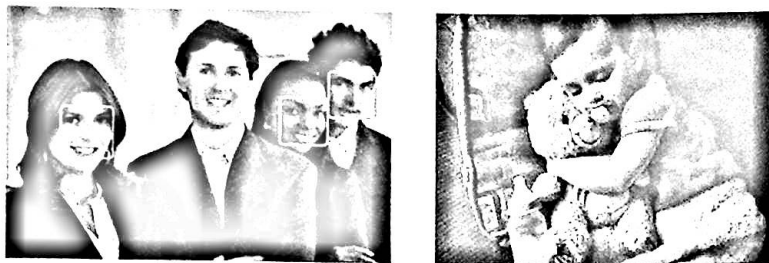


图 6-1 左图是人脸检测示例图。左图中算法能够自动找出人脸区域并用方框标记。右图显示了图像摘要算法示例。算法将根据输入图片自动产生描述语句：A young girl asleep on the sofa cuddling a stuffed bear(一个小女孩抱着一只毛绒玩具熊在沙发上睡觉)。

基于深度学习的图像摘要算法由 CNN 和 LSTM 两部分组成。其中，CNN 负责对图像特征进行提取，而 LSTM 负责实现图像与文字特征的转换翻译。CNN 提取的图像信息包括目标图像的类型、颜色、位置、类型等。在此基础上，LSTM 可以根据输入的图像特征转换单词信息并同时更新基于图像特征的状态信息。

本章以图像检测和图像摘要这两个问题为例，讲述使用 TensorFlow 处理 CNN 与 LSTM 相结合的模型及其应用。6.1 节介绍了 CNN 与 LSTM 相结合的 ReInspect 算法，并以基于 TensorFlow 的开源项目 TensorBox²⁸为基础，介绍 ReInspect 算法的实现。6.2 节介绍用于解决图像摘要问题的 CNN+LSTM 网络模型，并进一步说明 CNN+LSTM 的结合特性。另外，此例中还特别介绍了 TensorFlow 中使用 TFRecord 文件格式训练较大数据集的一般步骤。

6.1 CNN+LSTM 网络模型与图像检测问题

在前文第 4 章对 AlexNet、VGGNets、GoogLeNet、ResNets 等一系列经典 CNN

28 TensorBox 的 GitHub 网址：<https://github.com/TensorBox/TensorBox>。本章所介绍的 TensorBox 基于 TensorFlow 0.12 版本。因为 TensorFlow 仍在不断地推出新的版本，新的函数接口可能会不同于原先的接口函数。对此，TensorFlow 源码包提供了代码升级脚本 (tensorflow/tools/compatibility/tf_upgrade.py)，它可实现新版本接口函数的自动升级。只须运行 `python tf_upgrade.py --intree <源码文件夹顶级目录> -outtree <新生成的代码存放目录>` 即可。升级程序对于一些嵌套的参数赋值操作可能会更新失败，此时只须调试更改即可。

模型的介绍中曾提到它们一再刷新图像分类问题的纪录。与图像分类问题相比较,图像检测除了需要判断图像内容的类型之外,还需要标定图像中所有目标所在的位置。实际上,基于 CNN 的图像检测算法本质上依旧是解决一个目标类别的分类与位置坐标的回归预测问题。与基于深度学习的分类算法不同,检测算法在训练处理过程中都会使用深度网络中的某个卷积层的输出结果来表示图像的网格化特征提取结果。图像检测算法在计算损失函数时,输入到损失函数中的基本样本特征是网格子图特征。类似的,在检测算法的预测过程中,对目标类型判断和位置预测的基本样本特征同样也是网格子图特征。

这里介绍的 CNN+LSTM 网络结构主要用以解决遮挡目标的图像检测问题。OverFeat 算法是使用 CNN 模型处理图像检测问题的经典算法,曾在 ILSVRC 2013 中获得了图像检测问题的冠军,但在目标物体存在互相遮挡的情况下,识别效果并不好。ReInspect 算法是在 OverFeat 算法的基础上使用 LSTM 网络优化了遮挡目标的检测问题。本节将重点讲述 ReInspect 算法及其实现。

6.1.1 OverFeat 和 Faster R-CNN 图像检测算法介绍

通过第 4 章的介绍可以知道,深度卷积网络利用多层卷积结构,可以从图像中提取出物体的抽象特征。CNN 所提取的特征图是大小为 $W \times H \times C$ 的三维数据,其中 W 表示图像的宽度, H 表示高度, C 表示特征通道数。特征图的网格对应尺寸为 $1 \times 1 \times C$ 。以 VGGNets 或 GoogLeNet 为例,特征图中的每一个元素都是由原始图像对应位置的子图计算得出的。网格对应的原始子图计算区域称为感知域,同一网络中层次越深的卷积操作所得的特征图所对应的感知域越大,越趋向于目标物体的整体特征。一般来说,检测物体的大小应小于感知域的网格大小。

OverFeat 是使用 CNN 来解决图像检测问题的著名算法,诞生自纽约大学著名教授 Yann LeCun 所领导的实验室,并曾获得 ILSVRC2013 挑战赛图像检测问题的冠军。该算法对应的原始模型结构与 VGGNets 网络类似,它们都是由卷积层、池化层、激活层、全连接层构成的类似卷积网络模型。OverFeat 算法的最大贡献在于提出使用深度卷积网络模型一次完成分类、定位和检测三个计算机视觉任务。该算法利用深度卷积网络返回的卷积特征图实现图像网格的自动化分类的同时自动提取特征,最终以图像网格为样本判断目标类别以及预测目标相对于网格中心点的位置。

Faster R-CNN (Regions with CNN features) 算法²⁹同样是知名度较高的深度卷积神经网络图像检测算法。该方法是 2015 年 COCO 图像检测竞赛的基准比较算法。与 OverFeat 算法类似, 该方法同样使用 CNN 生成卷积网格特征图, 然后在扫描窗口中使用多个不同尺度、不同长宽比的锚点子窗口从卷积特征图中使用全连接层提取特征用于类别分类和位置预测。最后, 基于全连接特征的两个线性模型分别用于实现类别分类和位置预测。Faster R-CNN 的多尺度检测示意图如图 6-2 所示。这里强调的是, 与 OverFeat 算法相比, 而 Faster R-CNN 算法使用了多个锚点的子窗口提取卷积网格特征。当获取深度 CNN 网格特征后, Faster R-CNN 使用默认大小为 3×3 的扫描窗口获取卷积网格特征子图。然后, 将卷积网格特征子图的中心点与锚点子窗口对齐, 利用卷积网格特征子图插值生成不同尺寸、长宽比的锚点窗口大小的卷积网格特征子图。接着, 对于不同的锚点子窗口插值特征分别使用全连接层进一步将卷积特征统一转换为 512 维的特征。最后, 使用全连接线性分类器判定目标类别以及预测位置信息。

在 OverFeat 算法中, 仅对单个网格产生的卷积特征使用分类器判断目标类别以及预测位置信息。而 Faster R-CNN 算法则使用多尺度的锚点窗口融合网格卷积特征检测不同尺度与长宽比的图像目标, 如图 6-2 所示。一般而言, Faster R-CNN 算法的整体性能要优于 OverFeat 算法。

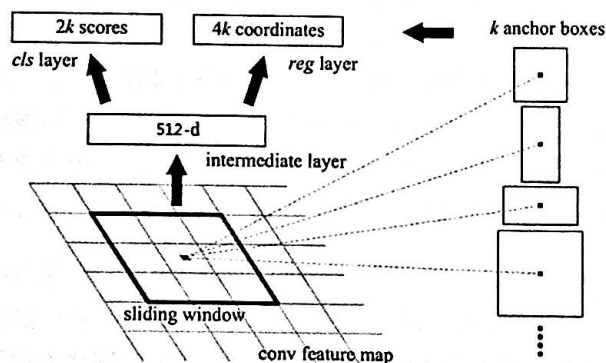


图 6-2 Faster R-CNN 算法中的多尺度检测示意图

²⁹ 论文原著为本章参考资料[15]。GitHub 上一个开源实现为: https://github.com/ShaoqingRen/faster_rcnn。

然而,无论是 OverFeat 算法又或是 Faster R-CNN 算法,它们最后都会对目标的检测结果的置信度进行排序,通过两两比较检测结果的重叠面积和置信度值,移除掉多余的检测结果。这种后处理方式在待检测目标结果的重叠区域非常小时是非常有效的。此时,即便检测算法在同一个目标区域返回了多个目标结果,也会根据它们的重叠面积而将其移除。而当检测目标存在遮挡时,这种处理方式极可能会移除掉被遮挡目标。若调整重叠面积判断参数,则又可能出现大量的重复检测结果。这就是遮挡目标检测问题。

6.1.2 遮挡目标图像检测方法

在 OverFeat 算法的基础上使用 LSTM 网络能够极大地改进遮挡目标的检测效果。在使用了 LSTM 神经网络后,一个网格卷积特征能够被复用,并产生多个序列特征以表征遮挡图像目标的检测结果。在算法实现方面,TensorBox 是一种基于 TensorFlow 的、可训练的图像检测识别框架,其中实现的 ReInspect 算法就是一种基于 CNN 与 LSTM 相结合的遮挡目标检测算法。

TensorBox 内部同时实现了 OverFeat 算法与 ReInspect 算法。它们的实现都使用 GoogLeNet 网络结构对图像信息划分网格进行编码,然后以网格卷积特征为基本单元预测网格卷积感知域内的目标及其位置信息。

在具体实现中,ReInspect 算法和 OverFeat 算法的不同之处还在于是否采用了 LSTM 神经网络和对应的损失函数的预处理操作。因为 ReInspect 算法采用 LSTM 网络结构对同一网格生成序列特征表征遮挡目标,这些特征序列的对应样本需要根据训练时的预测结果与真值进行匹配并赋值,以便计算损失函数。例如,网格产生的 m 个特征序列,对应的原始图像标记中有 n 个目标真值。在训练过程中, m 个样本序列对应的目标类型将根据 m 个样本的预测结果与 n 个目标真值进行相似匹配产生的。匹配的最大数量为 $\min(m, n)$,而没有匹配的特征序列则标记为负样本类型,对应的坐标位置填 0。另外,ReInspect 算法还采用了一种称为“缝合 (stitch)”的检测结果后处理方法。缝合后处理有两大特点,一个特点是同一网格产生的外包矩形检测结果不会进行比较移除,另一特点是单次缝合操作中,每一个检测结果最多只会移除一个重叠检测结果。这种处理能够确保同一网格卷积特征产生的重叠目标不会被认为是重复检测而被错误地移除。显然,当网格产生的特征序列长度为 1 时,缝合后处理操作

也就是经典的重复检测目标移除后处理算法。

由于 ReInspect 算法和 OverFeat 算法在整体流程上是基本一致的，故在基于 TensorBox 的内部实现中，两种算法仅依靠参数设置来区分。从实现的角度上来看，OverFeat 算法是 ReInspect 算法的简化版本。如表 6-1 所示，ReInspect 算法除了高层网络结构和损失函数预处理方法与 OverFeat 不同外，其他步骤这两者的实现是相同的。因此，在不引起混淆的情况下，本文的后续算法描述内容将以 ReInspect 算法为主。下一节将对各实现组成部分逐个详细说明。

表 6-1 ReInspect 算法与 OverFeat 算法在 TensorBox 内部实现中的异同

		ReInspect	OverFeat
训练过程	训练数据预处理	将图像划分为指定的网格大小，并以此为单位对目标进行预测	同左
	GoogLeNet 网络图像编码与感知域	使用已训练好的 GoogLeNet 网络对图像进行特征编码。最高层网络输出的网格大小与真值网格化的尺寸一致	同左
	LSTM 高层网络结构	对编码的网格特征使用 LSTM 网络生成高维特征	无此操作
	损失函数预处理操作	将特征序列的预测结果与真值进行匹配并赋予序列样本类型	无此操作
	损失函数	使用检测结果位置信息和置信度信息计算整体损失函数	同左
预测过程	检测后处理	使用全局匹配算法移除不同网格产生的冗余检测结果	根据检测置信度，移除重叠检测结果

6.1.3 ReInspect 算法实现及模块说明

在使用 TensorBox 前需要首先对其 C++实现的模块进行编译。若仅使用 OverFeat 算法，则只需要编译“检测后处理”模块。进入 TensorBox 项目所在目录，并在 Linux 终端运行以下 make 命令即可：

```
$ cd utils && make
```

若仅使用 ReInspect 算法则还需要编译“检测结果预处理”模块。此时需要将代

码中的 `hungarian` 模块完整地编译为动态链接库：

```
$ cd utils && make&& make hungarian
```

在准备完可运行程序后，还需要准备训练、验证和测试数据。运行源码目录下的 Linux 的脚本文件 `download_data.sh`，将下载 TensorFlow 提供的 GoogLeNet 模型参数、基于 OverFeat 算法的人脸检测模型以及人脸检测训练数据集。另外，TensorBox 使用 json 格式的样本标记格式。还可运行源码文件 `make_json.py`，对指定文件夹内的图像进行真值标记并保存为 json 格式。训练指定数据集的目标检测算法。例如：

```
python make_json.py train_images_dir train.json
```

上述命令会遍历文件夹 `train_images_dir` 中的所有图像文件，而后要求用手工方式标记目标在图像中的位置生成真值文件 `train.json`。`train.json` 文件中将含有目标类型和目标在图像中的位置信息。除了这种方式，还可以使用转换程序对其他格式图像检测真值数据进行格式转换，如 PASCAL 竞赛³⁰中的真值数据等。

配置运行参数后，运行 `train.py` 程序即可开启训练过程。训练完毕后，使用 `evaluate.py` 程序可以完成对数据集的样本测试。在 TensorBox 中，还提供了 iPython 版本的程序 `evaluate.ipynb`，可查看相关检测结果。

训练数据预处理

在前面的章节中我们提到用深度学习方法来解决图像检测问题，可以近似地认为一个分类与回归的组合问题。一方面需要通过分类判断目标是否存在，另一方面还需要通过回归预测目标的位置信息。在实现过程中，训练数据的预处理是问题转化的关键。图像检测问题中样本真值信息一般包含有目标的类型信息和目标在图像中的位置信息。训练数据预处理的目的是将单张图像中标记的真值信息转化为便于深度学习算法使用的格式。

TensorBox 使用图像网格化的方式来对数据真值信息进行转化。函数 `get_cell_grid` 根据输入的网格尺寸、网格行数和网格列数返回对应的网格信息列表。列表元素含有网格的左上角和右下角点坐标位置以及网格的序号。`AnnotationLib` 为位置信息类型，

30 PASCAL 竞赛主页：<http://host.robots.ox.ac.uk/pascal/VOC/>。

它含有一些矩形区域重叠面积的比较操作，代码如下。

```
import annolist.AnnotationLib as al
def get_cell_grid(cell_width, cell_height, region_size):
    cell_regions = []
    for iy in xrange(cell_height):
        for ix in xrange(cell_width):
            cidx = iy * cell_width + ix
            ox = (ix + 0.5) * region_size
            oy = (iy + 0.5) * region_size
            r = al.AnnoRect(ox - 0.5 * region_size, oy - 0.5 * region_size,
                           ox + 0.5 * region_size, oy + 0.5 * region_size)
            r.track_id = cidx
            cell_regions.append(r)
    return cell_regions
```

从 TensorBox 源码文件 `utils/data_utils.py` 中的函数 `annotation_to_h5` 可以获知具体转换方式。代码如下所示，算法首先获取配置文件中的网格尺寸，并验证网格尺寸对应网格行数与网格列数与原始图像大小的有效性。输入变量 `H` 表示读取的参数配置信息，变量 `a` 中加载了 json 图像真值文件的位置信息。调用 `get_cell_grid` 函数将图像的网格信息保存在 `cell_regions` 对象中。网格的数量为 `cells_per_image`。初始化 `box_list` 长度为 `cells_per_image` 的空的列表用于保存网格对应的真值目标信息。

```
def annotation_to_h5(H, a, cell_width, cell_height, max_len):
    region_size = H['region_size']
    assert H['region_size'] == H['image_height'] / H['grid_height']
    assert H['region_size'] == H['image_width'] / H['grid_width']
    cell_regions = get_cell_grid(cell_width, cell_height, region_size)

    cells_per_image = len(cell_regions)

    box_list = [[] for idx in range(cells_per_image)]
```

下面的循环负责遍历所有网格并判断网格与真值目标位置是否存在重叠。若存在位置重叠则将所有的真值信息以列表的形式保存至 `box_list` 对应的元素中。`intersection` 函数是 `annolist.AnnotationLib.AnnoRect` 的类成员函数。简单判断矩形对象是否存在重叠的代码如下。

```
for cidx, c in enumerate(cell_regions):
```

```
box_list[cidx] = [r for r in a.rects if all(r.intersection
(c))]
```

初始化样本标记 `boxes` 为(1, `cells_per_image`, 4, `max_len`, 1)尺寸大小的真值位置信息标记矩阵。尺寸向量中第一个 1 表示输入一张图像, `cells_per_image` 表示网格数量, 4 表示用于存储网格的中心点和长宽信息, 最后一个 1 表示目标的类型数值。`max_len` 表示允许的最大目标遮挡次数。在 OverFeat 算法中该值为 1, ReInspect 算法中该值对应 LSTM 网络的循环次数。类似的, `box_flags` 为(1, `cells_per_image`, 1, `max_len`, 1)尺寸大小的真值类型信息标记矩阵。

```
boxes = np.zeros((1, cells_per_image, 4, max_len, 1), dtype =
np.float)
box_flags = np.zeros((1, cells_per_image, 1, max_len, 1), dtype
= np.float)
```

因为 `TensorBox` 将网格作为基本单元来预测目标的位置以及实现目标类型的判断, 所以 `boxes` 保存的位置信息是目标真值在图像相对于对应网格的位置信息。更改 `box_list` 对应网格真值的位置为相对信息。考虑到最大重叠目标数量, 这里使用 `max_len` 进行了循环限制。通过比较目标真值位置中心和网格中心的距离和最大真值尺寸, 将满足条件的真值目标加入到 `unsorted_boxes` 队列中。满足条件的网格将作为正样本进行训练。否则, 将作为负样本进行训练。最后将真值目标根据与网格中心点的相对位置从小到大排序加入到 `boxes` 中。`box_flags` 保存对应的类型信息。因为 `box_flags` 值默认为 0, 因此限制有效目标的类型标签最小为 1, 代码如下所示。

```
for cidx in xrange(cells_per_image):          cell_ox = 0.5 *
(cell_regions[cidx].x1 + cell_regions[cidx].x2)
    cell_oy = 0.5 * (cell_regions[cidx].y1 + cell_regions[cidx].
y2)

    unsorted_boxes = []
    for bidx in xrange(min(len(box_list[cidx]), max_len)):
        # relative box position with respect to cell
        ox = 0.5 * (box_list[cidx][bidx].x1 + box_list[cidx]
[bidx].x2) - cell_ox
        oy = 0.5 * (box_list[cidx][bidx].y1 + box_list
[cidx][bidx].y2) - cell_oy

        width = abs(box_list[cidx][bidx].x2 - box_list[cidx]
[bidx].x1)
```

```

height= abs(box_list[cidx][bidx].y2 - box_list[cidx][bidx].y1)
        if (abs(ox) < H['focus_size'] * region_size and abs(oy)
< H['focus_size'] * region_size and
            width < H['biggest_box_px'] and height <
H['biggest_box_px']):
            unsorted_boxes.append(np.array([ox, oy, width,
height], dtype=np.float))
        for bidx, box in enumerate(sorted(unsorted_boxes, key=lambda
x: x[0]**2 + x[1]**2)):
            boxes[0, cidx, :, bidx, 0] = box
            box_flags[0, cidx, 0, bidx, 0] = max(box_list[cidx][bidx].silhouetteID, 1)

    return boxes, box_flags

```

简而言之，以上转换的目的就是将图像分解为若干个大小相等的子图块，并对每一个子图块赋予检测目标类型和目标位置信息。在上述转换算法中目标最大遮挡次数 `max_len` 是区分 ReInspect 算法和 OverFeat 算法的关键。在 ReInspect 算法中 `max_len > 1`，高层网络使用了基于 LSTM 的循环神经网络，允许同一个网格产生多个存在遮挡的检测结果。而 OverFeat 算法的网络高层仅使用了线性分类器，一个网格只能有一个检测结果，此时 `max_len` 仅允许为 1。

队列是 TensorFlow 为训练数据快速 I/O 操作提供了一种独立线程异步运行机制。这种机制的典型应用就是多个线程准备训练数据进行入队列操作，与此同时，训练程序则在另一个线程中运行优化并通过调用出队列操作来获取训练数据。TensorFlow 中的队列类型主要是指 `FIFOQueue` 和 `RandomShuffleQueue` 这两种。本例中使用的是 `FIFOQueue` 队列。在 TensorFlow 的官方文档³¹中可以查阅相关介绍。在下一节图像摘要的例子中将介绍使用 `RandomShuffleQueue` 队列进行大数据训练。定义好线程队列后，可使用 `tf.train.start_queue_runners` 函数开启会话线程队列机制，并在独立线程中运行会话的数据入队列操作。TensorFlow 内部将自动实现数据入/出队列线程和训练线程的同步等待。

算法采用线程队列的方式为训练数据提供数据。如下代码所示，令输入图像数据

31 介绍线程和队列的官方文档网址：https://www.tensorflow.org/programmers_guide/threading_and_queues。

表示为 `x_in`、对应的网格样本置信度表示为 `confs_in`、`boxes_in` 表示预测位置坐标信息。列表对象 `q` 表示 TensorFlow 内部容量上限为 30 的先进先出队列。队列中的每个元素是由 `x_in`、`confs_in` 和 `boxes_in` 构成的元组。`enqueue_op` 表示入队列操作。

```
x_in = tf.placeholder(tf.float32)
confs_in = tf.placeholder(tf.float32)
boxes_in = tf.placeholder(tf.float32)
q = {}
enqueue_op = {}
for phase in ['train', 'test']:
    dtypes = [tf.float32, tf.float32, tf.float32]
    grid_size = H['grid_width'] * H['grid_height']
    shapes = (
        [H['image_height'], H['image_width'], 3],
        [grid_size, H['rnn_len'], H['num_classes']],
        [grid_size, H['rnn_len'], 4],
    )
    q[phase] = tf.FIFOQueue(capacity=30, dtypes=dtypes,
        shapes=shapes)
    enqueue_op[phase] = q[phase].enqueue((x_in, confs_in,
        boxes_in))
```

函数 `make_feed` 负责为 TensorFlow 网络提供占位符的输入信息。函数 `thread_loop` 实现了网络训练数据的入队列操作。

```
def make_feed(d):
    return {x_in: d['image'], confs_in: d['confs'], boxes_in:
        d['boxes'],
            learning_rate: H['solver']['learning_rate']}

def thread_loop(sess, enqueue_op, phase, gen):
    for d in gen:
        sess.run(enqueue_op[phase], feed_dict=make_feed(d))
```

在相同的 Session 作用域下执行入队列操作。随后开启入队列线程。其中对象 `gen` 表示 for 循环 yield 迭代器。最后，令入队列线程为守护线程。确保直至整个进程结束时才退出循环。其中 `config` 由函数 `tf.ConfigProto` 返回，用以指示 TensorFlow 是在 CPU 或 GPU 中运行。函数 `tf.train.start_queue_runners` 开启运行会话中的线程队列。

```
with tf.Session(config=config) as sess:
    tf.train.start_queue_runners(sess=sess)
```

```

for phase in ['train', 'test']:
    # enqueue once manually to avoid thread start delay
    .....
    d = gen.next()
    sess.run(enqueue_op[phase], feed_dict=make_feed(d))
    t = threading.Thread(target=thread_loop,
                        args=(sess, enqueue_op, phase, gen))
    t.daemon = True
    t.start()

```

GoogLeNet 网格图像编码与感知域

对图像真值进行转换后还需要对图像块进行特征提取,而后才能转换为一般意义上的深度学习分类与回归问题。在前面的章节中已经介绍了 GoogLeNet 网络结构与 VGGNet 网络结构。这两种网络结构都使用了池化层对原图进行降采样处理。TensorBox 默认使用的网格尺寸为 32×32 ,即进行过 5 次 $[2,2]$ 大小的降采样操作后的网络输出结果可等价视为以网格为单位的特征提取结果。

相较于 VGGNet 网络结构,GoogLeNet 使用 1×1 卷积等并行结构,同时扩展了网络的宽度和深度。按照 TensorFlow 的网络定义经过相同数量的降采样操作后,GoogLeNet 顶层网络的单个网格特征拥有更大的感知域。

TensorFlow 官方源码中,给出了使用 TF-Slim 实现的 GoogLeNet 和 VGGNet 结构,其结构如表 6-2 所示。TensorBox 默认使用 GoogLeNet 作为训练模型的使用方法。训练时,通过 slim 模块的 assign_from_checkpoint_fn 函数实现先验模型参数的加载。关键代码如下所示,slim.assign_from_checkpoint_fn 函数有两个输入参数,GoogLeNet 其中一个参数用于指定 GoogLeNet 网络结构的模型存档地址,另一个参数用于指定需要训练的变量集合。

```

init_fn = slim.assign_from_checkpoint_fn(
    '%s/data/inception.ckpt'%os.path.dirname(os.path.realpath(__file__)),
    [x for x in tf.all_variables() if x.name.startswith('InceptionV1')
     and not H['solver']['opt'] in x.name])

```


表 6-2 GoogLeNet 和 VGG19 的网格各层感知域大小

GoogLeNet					VGGNet19			
层数	名称	类型	步长	感知域	名称	类型	步长	感知域
1	Conv2d_1a_7x7	卷积[7,7]	2	[7,7]	conv1_1	卷积[3,3]	1	[3,3]
2	MaxPool_2a_3x3	池化[3,3]	2	[11,11]	conv1_2	卷积[3,3]	1	[5,5]
3	Conv2d_2b_1x1	卷积[1,1]	1	[11,11]	pool1	池化[2,2]	2	[6,6]
4	Conv2d_2c_3x3	卷积[3,3]	1	[19,19]	conv2_1	卷积[3,3]	1	[10,10]
5	MaxPool_3a_3x3	池化[3,3]	2	[27,27]	conv2_2	卷积[3,3]	1	[14,14]
6	Mixed_3b	inception	1	[43,43]	pool2	池化[2,2]	2	[16,16]
7					conv3_1	卷积[3,3]	1	[24,24]
8	Mixed_3c	inception	1	[59,59]	conv3_2	卷积[3,3]	1	[32,32]
9					conv3_3	卷积[3,3]	1	[40,40]
10	MaxPool_4a_3x3	池化	2	[75,75]	conv3_4	卷积[3,3]	1	[48,48]
11	Mixed_4b	inception	1	[107,107]	pool3	池化[2,2]	2	[52,52]
12					conv4_1	卷积[3,3]	1	[68,68]
13	Mixed_4c	inception	1	[139,139]	conv4_2	卷积[3,3]	1	[84,84]
14					conv4_3	卷积[3,3]	1	[100,100]
15	Mixed_4d	inception	1	[171,171]	conv4_4	卷积[3,3]	1	[116,116]
16					pool4	池化[2,2]	2	[124,124]
17	Mixed_4e	inception	1	[203,203]	conv5_1	卷积[3,3]	1	[156,156]
18					conv5_2	卷积[3,3]	1	[188,188]
19	Mixed_4f	inception	1	[235,235]	conv5_3	卷积[3,3]	1	[220,220]
20					conv5_4	卷积[3,3]	1	[252,252]
21	MaxPool_5a_2x2	池化[2,2]	2	[251,251]	pool5	池化[2,2]	2	[268,268]
22	Mixed_5b	inception	1	[315,315]	fc6	卷积[7,7]	1	[460,460]
23					fc7	卷积[1,1]	1	[460,460]
24	Mixed_5c	inception	1	[379,379]	fc8	卷积[1,1]	1	[460,460]
25								

表 6-2 中列举出了 TensorBox 中 GoogLeNet 和 VGG19³² 网络结构的所有卷积层和池化层, 以及对应的感知域。其中, GoogLeNet 网络中的 Inception 网络结构在前面介绍 CNN 的章节中已经介绍过。Inception 网络结构包括 4 个并行的网络分支, 分别使用 1×1 卷积、 3×3 卷积、 5×5 卷积和池化分支构成。因 Inception 网络结构中的分支存在两层卷积, 所以一个 Inception 网络结构会令网络深度加 2。感知域是指高层网络中的单个网格特征对应原始图像输入端的有效计算区间。这里以 GoogLeNet 网络结构的前 4 层为例进行说明。输入层经过一个卷积核大小为 7×7 、步长为 2 的卷积操作到达第一层网络。显然第一层网络中的网格特征感知域为 $[7, 7]$ 。而后, 第一层网络经过一个尺寸为 3×3 、步长为 2 的池化操作进入到第三层网络。因为池化尺寸为 $[3, 3]$, 所以感知区间相较于上层增加了 $[2, 2]$ 个单元。又因为此时相较于输入层的计算步长为 2, 总体上感知域尺寸增加了 $[2 \times 2, 2 \times 2]$ 个单元。此时的感知域为 $[11, 11]$, 相较于输入层的计算步长为 4。第三层网络操作是大小为 1×1 、步长为 1 的单元卷积, 此时感知域尺寸保持不变。第四层网络操作是大小为 3×3 、步长为 1 的卷积, 此时感知域尺寸增加量为 $[2 \times 4, 2 \times 4]$, 感知域变为 $[19, 19]$ 。余下层次对应的感知域依次类推。

从表 6-2 中可以看出 GoogLeNet 的最大感知域为 $[379, 379]$, TensorFlow 实现中对应层的名称是 Mixed_5c。而 TensorBox 默认的实现加载的是 Mixed_5b, 对应的感知域大小是 $[315, 315]$ 。另一方面, 因为 VGGNet 中的 fc6 层使用了非填充式卷积, 从这一层开始的网格特征与 TensorBox 提供的样本网格标记方式不再一一对应。所以在不改动 TensorBox 样本网格标记的前提下, VGG19 单个网格的最大有效感知域是 $[268, 268]$, 对应的层次是 pool5。

前文提到了图像检测问题可以视为一个图像网格化特征的目标分类和位置信息回归预测的组合问题。原始的 json 信息是以图像的左上角点为原点给出的。经过真值样本图像网格转化后, 每一个图像网格被赋予真值信息。真值信息包括网格对应待检测目标类型和待检测目标的位置信息。其中, 待检测目标的位置信息是以网格中心为原点给出的。

当网格待检测目标的位置信息尺寸远大于网格的感知域时, 这意味着网格抽取的

³² vgg 网络包含两个不同的实现版本 vgg_16 和 vgg_19。其中 16 和 19 分别代表卷积层的数量。

特征是待检测目标的部分外观特征,而并非待检测目标的全部特征。当这种类型的样本在训练集中占比较大时,会对算法性能产生较大的不良影响,可能会导致过拟合或者不收敛。另一方面,当网格的待检测真值尺寸信息远小于网格特征的感知域时,这意味着感知域中的大部分信息都是非目标区域。此时,网格特征中含有较多的非目标特征信息,这可能需要更多的训练样本进行参数调整防止模型的过拟合。若目标尺寸过小且随机出现在网格中,则可能无法抽取目标的有效特征,以致产生训练不收敛的结果。因此,对不同的检测目标集合选择不同的感知域是很有必要的。与 VGG19 相比较,GoogLeNet 网格具有更好的宽度和深度,能够更好地反映网格内感知域内的目标特征。

TensorBox 的内部实现提供了放大分辨率的网格特征选项。可在 TensorBox 源码文件 `train.py` 中找到 `rezoom` 函数。相关参数设置可在 `hypes` 文件夹下的 `json` 文件中设置。TensorBox 默认有 `lstm.json`、`lstm_rezoom.json`、`overfeat.json`、`overfeat_rezoom.json` 四种配置文件。其中, `lstm_rezoom.json` 和 `lstm.json` 分别表示 ReInspect 算法下使用和不使用低层网格特征的参数配置。`overfeat_rezoom.json` 和 `overfeat.json` 分别表示 OverFeat 算法下使用和不使用低层网格特征的参数配置。其中 `overfeat_rezoom.json` 格式的配置文件中的“`use_rezoom`”:true,表示使用低层网络特征。

当使用放大分辨率的网格特征时,系统将使用 `rezoom` 函数。该函数先通过输入高层网格特征预测的目标位置定位 4 个中心点距离最近的低层网格。然后,再利用两者的距离对 4 个低层网格特征进行线性插值获取融合特征。此时,系统将同时利用插值融合网格特征和高层网格特征进行目标类型的分类训练和位置的回归预测训练。由 `utils/googlenet_load.py` 文件中可以看到, TensorBox 默认加载的是 GoogLeNet 模型。使用的高层特征名称是 `Mixed_5b`,对应的 4 个低层特征对应的网络层名称是 `Mixed_3b`。在 `json` 配置文件中的 `rezoom_w_coords` 和 `rezoom_h_coords` 参数项分别含有两个值,用来表示 4 个左上、右上、左下、右下 4 个插值网格相较于高层网格特征预测的目标中心点的网格相对距离。最后将与 `region_size` 结合确认具体的 4 个低层插值网格。

LSTM 高层网络结构

ReInspect 算法在 GoogLeNet 网格上继续添加 LSTM 循环网络结构抽取特征。在

util/*.json 格式的配置文件中 use_lstm 的值为 true 时,表示使用 LSTM 高层网络。rnn_len 表示堆叠的 LSTM 循环网络次数。前文介绍的训练数据预处理中的网格内最大覆盖目标数量与 rnn_len 的值相同。ReInspect 算法使用 LSTM 网络的中心思想是使用 LSTM 循环网络对 GoogLeNet 网络特征进行多次特征提取。若网格存在被遮挡的目标,ReInspect 算法则期望能够利用 LSTM 网络的记忆特性对被遮挡的目标做进一步的判断检测。令网格样本真值中存在 $n(n \leq \text{rnn_len})$ 个目标真值,对于生成的 rnn_len 个 LSTM 循环网络特征,可以设置不同的真值映射规则。与真值匹配的序列样本被设置为正样本,其他样本被设置为负样本。这种方式将遮挡目标检测问题转化为一种目标序列的分类问题。

关键代码如下所示,其中 H['lstm_size']表示 LSTM 网络的特征维数。配置参数 H['num_lstm_layers']表示 LSTM 网络的叠加次数。函数 rnn_cell.MultiRNNCell 负责实现 LSTM 的多次叠加。叠加的 LSTM 网络使用不同的参数。输入参数[lstm_cell] * H['num_lstm_layers']为 LSTM 网络定义操作的列表对象。理论上叠加的次数越多,能概括的图像遮挡情况越复杂,但需要的样本也越多。

```
def build_lstm_inner(H, lstm_input):
    '''
        build lstm decoder
    '''
    lstm_cell = rnn_cell.BasicLSTMCell(H['lstm_size'], forget_bias=0.0,
    state_is_tuple=False)
    if H['num_lstm_layers'] > 1:
        lstm = rnn_cell.MultiRNNCell([lstm_cell] *
    H['num_lstm_layers'], state_is_tuple=False)
    else:
        lstm = lstm_cell
```

输入数据根据图像网格大小和同时输入的图像数量划分为大小为 batch_size 的二维数据包。如前文所述,假设网格中遮挡目标的最大重叠次数为 H['rnn_len'],则可在 for 循环中结合条件语句使用 tf.get_variable_scope().reuse_variables()实现 lstm 网络参数的复用。若不使用 tf.get_variable_scope().reuse_variables()函数,在变量作用域'RNN'中的 for 循环中第二次使用 lstm(lstm_input, state)函数时,将创建两个同名的张量产生运行时错误。不使用 tf.get_variable_scope().reuse_variables()函数时,不允许返回两个 name 属性相同的张量。变量作用域是张量 name 属性的组成部分。

最后, 基于网格的 batch 数据将输入 lstm 网络, 产生 $H['rnn_len']$ 次特征提取结果以反映遮挡目标的图像特征 output。关于 reuse_variables 函数及其相关概念, 可参见官方文档³³。

```
batch_size = H['batch_size'] * H['grid_height'] * H['grid_
width']
state = tf.zeros([batch_size, lstm.state_size])
outputs = []
with tf.variable_scope('RNN', initializer=tf.random_uniform_
initializer(-0.1, 0.1)):
    for time_step in range(H['rnn_len']):
        if time_step > 0: tf.get_variable_scope().reuse_variables()
        output, state = lstm(lstm_input, state)
        outputs.append(output)
    return outputs
```

由此可见, 在 ReInspect 算法中一个网格特征最多能够产生 “rnn_len” 个目标检测结果。而 OverFeat 算法中 1 个网格特征最多只能有 1 个检测结果。这两种算法的样本预处理是不同的。训练时 ReInspect 算法前向传播, 单个网格将会返回多个特征和多个类型标签。在计算损失函数前网格 ReInspect 算法需要将序列特征的预测结果与真值进行匹配后, 再对序列样本赋予类别标签计算损失函数。所以在损失函数的计算前, 有必要对样本序列的排列次序进行调整。这项操作被称为损失函数预处理。

损失函数预处理

通过调整循环 LSTM 网络结构产出特征的标签信息, 能够确保特征序列能够正确反映遮挡检测目标。例如, 根据前文对训练数据预处理的介绍, 当循环 LSTM 网络的循环次数为 m 时。每一个网格将产出 m 对特征样本和标签信息组成的序列。在训练样本预处理过程中, 我们已经确保了检测目标的真值信息优先出现在序列的前面。但, LSTM 生成的序列与真值之间并无关联。若网格对应存在 n 个遮挡目标时, 前 n 个网格特征序列对应的具体真值信息是没有逻辑关联的。认定第一个序列特征对应第一个真值是不合理的, 而没有确定特征与真值的匹配关系将无法计算损失函数。因此, 在训练过程中应考虑特征与真值信息的匹配关系。显然, 这是一个典型的二部图问题。ReInspect 算法使用匈牙利算法完成前 n 个特征序列与真值之间的匹配, 即希望训练

33 https://www.tensorflow.org/programmers_guide/variable_scope。

时前 n 个特征序列的预测结果与真值的总体加权匹配损失是最小的。匈牙利算法³⁴早在 1965 年由匈牙利数学家 Edmonds 提出,是解决二部图问题的经典算法,限于篇幅,本书不再复述。

训练时 ReInspect 算法会对网格特征序列的预测结果与真值信息进行匹配调整预测结果,匹配问题如图 6-3 所示。

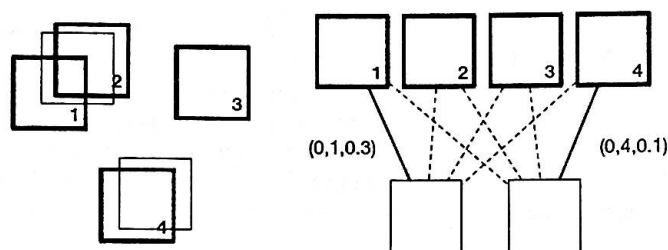


图 6-3 检测结果匹配预处理示意图

假设网格最多产生 4 个特征序列预测结果由数字标出,与该网格对应的目标真值数量为 2。通过定义一个 $[2,4]$ 大小的代价矩阵,从中挑选出 2 个不同行列的元素求和用以表示匹配的代价值。匹配问题就是找出真值信息与检测结果的最小代价对应关系。匹配距离计算方法如下公式所示。

$$\Delta(b_i, \tilde{b}_j) = (o_{ij}, r_j, d_{ij})$$

其中, b_i 表示真值位置, \tilde{b}_j 表示特征序列的一个预测位置。 $o_{ij} \in \{0,1\}$, 当 b_i 与 \tilde{b}_j 的重叠区域比例大于阈值时 $o_{ij} = 0$, 否则 $o_{ij} = 1$ 。 r_j 表示特征在序列中的序号, 越靠前的预测结果匹配代价越小。 d_{ij} 表示 b_i 与 \tilde{b}_j 对应区域中心点的距离, 距离越小代价越小。代价函数优先比较 o_{ij} , 其次比较 r_j , 最后考量 d_{ij} 。匹配距离的计算方式如图 6-3 所示序列特征 1 和 4 对应的匹配距离为 $(0,1,0.3)$ 和 $(0,4,0.1)$ 。这种元组信息在程序实现中将以加权方式转化为统一量纲的距离值。

训练时, ReInspect 算法包括 3 种不同的匹配方式。一种是如图 6-3 所示的匈牙利算法匹配方式 L_{hung} , 该方法下 rnn_len 个循环 LSTM 网络特征预测结果将与 n 个真

34 匈牙利算法介绍: https://en.wikipedia.org/wiki/Hungarian_algorithm。

值进行匹配, 匹配成功的 n 个网格特征将被赋予目标真值剩下的 $\text{rnn_len}-n$ 个网格特征将作为负样本进行训练。第二种方法记为 L_{first} , 是将 n 个真值与特征序列中的前 n 预测结果进行匈牙利匹配, 对前 n 个特征序列按照匹配结果赋予真值。第三种方法是固定检测结果与真值之前的关系记为 L_{fix} 。经过上述任一方式转换后, 都可以对循环 LSTM 网络特征进行一般意义上的样本训练。

TensorBox 中的 ReInspect 算法是按照 L_{hung} 方法对检测结果进行预处理的。在 `train.py` 文件中的 `build_forward_backward` 函数可以看到对 TensorFlow 自定义动态库的调用方法。在源码文件 `utils\hungarian\hungarian.cc` 查看具体的逻辑实现代码。TensorFlow 提供了 C/C++ 模块扩展接口, 在其官方文档³⁵中也可以查看自定义算子的详细说明和示例。

损失函数

如同一般的深度学习问题, 图像检测算法需要输入大量的训练图像数据通过损失函数计算的反向传播梯度实现模型参数优化, 并最终训练得出检测模型。图像检测问题不仅需要判断目标的类型, 还需要得出目标的位置信息。因此, 这一问题实质上是一个关于图像的分类问题和位置信息的逻辑回归问题的组合问题。在实际应用中, 可以使用线性加权的方式来对检测目标的类型和位置信息进行融合加权。TensorBox 所使用的损失函数如下式所示。

$$L(G, C, f) = \sum_{i \in N} \left(\alpha \sum_{i=1}^{|G|} \frac{l_{\text{pos}}(b_{\text{pos}}^i, \tilde{b}_{\text{pos}}^{f(i)})}{|G|} + \beta \sum_{j=1}^{|C|} \frac{l_c(\tilde{b}_c^j, y_j)}{|C|} \right)$$

其中, N 表示图像数量。集合 G 表示单张图像标记的目标位置信息集合。网格位置信息数量 $|G|$ 由原始标注信息与覆盖网格的数量决定。集合 C 表示划分的图像网格集合, 对于相同尺寸的图像, 数值 $|C|$ 是保持不变的。 l_{pos} 表示目标位置信息损失函数。它反映网格标记目标位置坐标值与预测坐标值的差异。在程序中该函数是位置向量差的绝对值之和。它有两个输入参数, 其中 b_{pos}^i 表示第 i 个网格真值标记的位置区域信息。 $\tilde{b}_{\text{pos}}^{f(i)}$ 表示与对应网格特征的位置预测结果。 $f(i)$ 用于表示第 i 个真值位置信

35 自定义算子文档: https://www.tensorflow.org/extend/adding_an_op。

息对应的网格区域映射关系。在程序中 $f(i)$ 由区域的中心点距离决定。 l_c 表示网格目标分类信息损失函数，用于度量网格类型与目标类型的一致性。它的两个输入参数， \tilde{b}_c^j 表示网格 j 特征的分类判断置信度张量。 y_j 表示 $[0, 1]$ 类型标签张量。因每张图像的真值标记区域 G 数量不同需要除以该值进行归一化。参数 α 和参数 β 用于调整位置信息和分类信息损失函数的占比权重。

$$\begin{aligned}
 l_{\text{pos}}(b_{\text{pos}}^i, \tilde{b}_{\text{pos}}^{f(i)}) &= \text{sum}(\text{abs}(b_{\text{pos}}^i - \tilde{b}_{\text{pos}}^{f(i)})) \\
 l_c(\tilde{b}_c^j, y_j) &= \text{sum} \left(-y_i \cdot \log \frac{e^{\tilde{b}_c^j - \max(\tilde{b}_c^j)}}{\text{sum}(e^{\tilde{b}_c^j - \max(\tilde{b}_c^j)})} \right) \\
 &= \text{sum} \left(-y_i \cdot \left(\tilde{b}_c^j - \max(\tilde{b}_c^j) - \log \left(\text{sum}(e^{\tilde{b}_c^j - \max(\tilde{b}_c^j)}) \right) \right) \right)
 \end{aligned}$$

位置信息和分类信息损失函数的值分别在不同的量纲范畴表示，这里使用的参数 α 和参数 β 为经验值。默认情况 $\alpha = 0.1$ ， $\beta = 1$ ，即强调目标区域的分类判断对最终检测结果的影响程度是位置信息误差的 10 倍。从 TensorBox 源码文件 `train.py` 中的 `build_forward_backward` 函数可知损失函数中的分类信息由张量 `confidences_loss` 表示，位置信息由张量 `boxes_loss` 表示。其中，`confidences_loss` 是由分类置信度带入到函数 `tf.nn.sparse_softmax_cross_entropy_with_logits` 获取的。通过查找 TensorFlow 的源码关键字，可在文件 `tensorflow\core\kernels\xent_op.h` 了解到其具体实现如以上公式所示。

在程序实现中，初始时图像编码神经网络会填入预先训练好的模型参数，高层自定义网络层次使用均值为 0 且方差较小的初始权，以此确保网络在初始训练时是易于优化的，不易陷入局部最优解中。

TensorBox 中并没有采用一些非均衡数据的优化处理方法。而在 Faster R-CNN 的训练过程中对 `batch` 包的数据正负样本比例进行了 1:1 调整。这避免了负样本类型过多造成的非均衡数据训练问题。因后续实验中采用的实验数据正样本几乎存在于每张图像中，所以不存在较严重的非均衡数据问题。当使用 TensorBox 训练一些正样本标记较少的样本时，应考虑添加非均衡数据的优化操作。

检测后处理

ReInspect 算法使用循环 LSTM 网络结构令单个网格单元产生多个序列特征，并以此特征序列对遮挡目标进行检测。然而，相邻的网格特征因感知域范围存在重叠部分，直接使用样本预处理获取的网格标签训练循环 LSTM 网络同样会产生冗余的检测结果。即使利用 LSTM 网络特征获取了遮挡目标检测结果，仍然可能存在冗余的检测结果。如果使用传统的置信度排序与重叠判断等后处理方法来去除冗余检测，那么利用 LSTM 网络特征检测获取的遮挡目标将同样被认为是冗余检测而被移除。因此，一方面我们期望相同网格产生的 LSTM 网络特征能够正确地反映遮挡目标的特征信息，另一方面我们同样期望能够移除相邻网格特征产生的冗余检测结果。针对网格检测结果分组的重复检测移除后处理算法很好地解决了这一问题。

虽然使用网格序列来检测遮挡目标能够较好地提升遮挡目标的检测率。但是，相邻网格的感知域是存在重叠部分的，也就是说相邻网格的检测结果是存在目标重复检测的。因此，网格检测完毕后还需要后处理算法移除网格间的重复检测结果，同时还要保留遮挡目标的检测结果。ReInspect 算法使用的后处理是基于网格进行处理的。如图 6-4 所示，若两重叠检测结果来自相同网格，则同时保留。若两重叠检测结果来自不同网格，则移除可信度较小的检测结果。

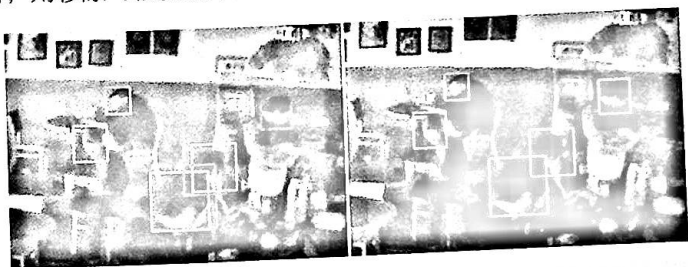


图 6-4 ReInspect 算法后处理示意图

另外，为了保留重叠目标检测结果，单次移除操作中一个网格检测结果最多只能移除一个来自另一个网格的重叠检测结果。考虑到一个检测结果可能会与多个不同网格的检测结果重叠，检测后处理方法再一次使用了匈牙利算法实现了不同网格重复检测结果的移除。考虑到同一目标可能出现多次重复检测，后处理算法采用了置信度分段的方式对检测结果进行了多次匹配移除。

息对应的网格区域映射关系。在程序中 $f(i)$ 由区域的中心点距离决定。 l_c 表示网格目标分类信息损失函数，用于度量网格类型与目标类型的一致性。它的两个输入参数， \tilde{b}_c^j 表示网格 j 特征的分类判断置信度张量。 y_j 表示 $[0, 1]$ 类型标签张量。因每张图像的真值标记区域 G 数量不同需要除以该值进行归一化。参数 α 和参数 β 用于调整位置信息和分类信息损失函数的占比权重。

$$l_{\text{pos}}(b_{\text{pos}}^i, \tilde{b}_{\text{pos}}^{f(i)}) = \text{sum}(\text{abs}(b_{\text{pos}}^i - \tilde{b}_{\text{pos}}^{f(i)}))$$

$$l_c(\tilde{b}_c^j, y_j) = \text{sum}\left(-y_i \cdot \log \frac{e^{\tilde{b}_c^j - \max(\tilde{b}_c^j)}}{\text{sum}(e^{\tilde{b}_c^j - \max(\tilde{b}_c^j)})}\right)$$

$$= \text{sum}\left(-y_i \cdot \left(\tilde{b}_c^j - \max(\tilde{b}_c^j) - \log\left(\text{sum}(e^{\tilde{b}_c^j - \max(\tilde{b}_c^j)})\right)\right)\right)$$

位置信息和分类信息损失函数的值分别在不同的量纲范畴表示，这里使用的参数 α 和参数 β 为经验值。默认情况 $\alpha = 0.1$ ， $\beta = 1$ ，即强调目标区域的分类判断对最终检测结果的影响程度是位置信息误差的 10 倍。从 TensorBox 源码文件 `train.py` 中的 `build_forward_backward` 函数可知损失函数中的分类信息由张量 `confidences_loss` 表示，位置信息由张量 `boxes_loss` 表示。其中，`confidences_loss` 是由分类置信度带入到函数 `tf.nn.sparse_softmax_cross_entropy_with_logits` 获取的。通过查找 TensorFlow 的源码关键字，可在文件 `tensorflow\core\kernels\xent_op.h` 了解到其具体实现如以上公式所示。

在程序实现中，初始时图像编码神经网络会填入预先训练好的模型参数，高层自定义网络层次使用均值为 0 且方差较小的初始权，以此确保网络在初始训练时是易于优化的，不易陷入局部最优解中。

TensorBox 中并没有采用一些非均衡数据的优化处理方法。而在 Faster R-CNN 的训练过程中对 `batch` 包的数据正负样本比例进行了 1:1 调整。这避免了负样本类型过多造成的非均衡数据训练问题。因后续实验中采用的实验数据正样本几乎存在于每张图像中，所以不存在较严重的非均衡数据问题。当使用 TensorBox 训练一些正样本标记较少的样本时，应考虑添加非均衡数据的优化操作。

检测后处理

ReInspect 算法使用循环 LSTM 网络结构令单个网格单元产生多个序列特征，并以此特征序列对遮挡目标进行检测。然而，相邻的网格特征因感知域范围存在重叠部分，直接使用样本预处理获取的网格标签训练循环 LSTM 网络同样会产生冗余的检测结果。即使利用 LSTM 网络特征获取了遮挡目标检测结果，仍然可能存在冗余的检测结果。如果使用传统的置信度排序与重叠判断等后处理方法来去除冗余检测，那么利用 LSTM 网络特征检测获取的遮挡目标将同样被认为是冗余检测而被移除。因此，一方面我们期望相同网格产生的 LSTM 网络特征能够正确地反映遮挡目标的特征信息，另一方面我们同样期望能够移除相邻网格特征产生的冗余检测结果。针对网格检测结果分组的重复检测移除后处理算法很好地解决了这一问题。

虽然使用网格序列来检测遮挡目标能够较好地提升遮挡目标的检测率。但是，相邻网格的感知域是存在重叠部分的，也就是说相邻网格的检测结果是存在目标重复检测的。因此，网格检测完毕后还需要后处理算法移除网格间的重复检测结果，同时还要保留遮挡目标的检测结果。ReInspect 算法使用的后处理是基于网格进行处理的。如图 6-4 所示，若两重叠检测结果来自相同网格，则同时保留。若两重叠检测结果来自不同网格，则移除可信度较小的检测结果。

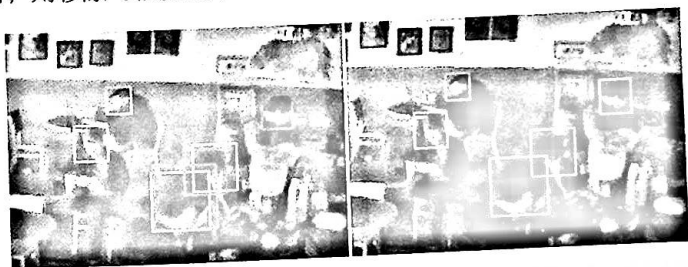


图 6-4 ReInspect 算法后处理示意图

另外，为了保留重叠目标检测结果，单次移除操作中一个网格检测结果最多只能移除一个来自另一个网格的重叠检测结果。考虑到一个检测结果可能会与多个不同网格的检测结果重叠，检测后处理方法再一次使用了匈牙利算法实现了不同网格重复检测结果的移除。考虑到同一目标可能出现多次重复检测，后处理算法采用了置信度分段的方式对检测结果进行了多次匹配移除。

关于检测后处理方面的具体实现,可以通过查看 TensorBox 源码文件 `utils/stitch_wrapper.pyx` 做进一步了解。从函数 `stitch_rects` 中的列表变量 `thresholds` 可查看置信度分段的含义。每一个列表元素由一个二元的元组组成。元组的两个值分别表示当前参与匹配移除的检测目标最小置信度,另一个表示确保加入到检测结果的最大置信度值。使用这种后处理方式,能够有效地移除置信度差异较大的重复检测,同时亦可保留置信度相当的遮挡目标。

6.1.4 ReInspect 算法的实验数据与结论

评估标准与实验数据

ReInspect 论文原文中使用了两个数据集进行对比实验。其中一个数据集是作者使用固定网络摄像机拍摄的室内场景剪辑生成。该数据集是一个较密集人群组成的室内场景,名为 Brainwash。以视频的方式可以收集相当数量的图像数据。使用这种方式能够确保算法的实现不会受到训练数据集过小的限制。另一个实验数据集使用的是公开数据集 TUD-Crossing³⁶。对比实验使用相同的参数在这两个不同的数据集上进行测试。

Brainwash 数据集中收集了 11917 张共计标记了 91146 个人头样本的真值图像。收集的图像数据是以 100 秒为间隔从视频中抽取的。测试集和验证集图像各占有 1000 张,余下的图像被用于训练。训练集与测试集是不存在同样样本的。整个训练集合共有 82906 个检测目标样本实例。测试集和验证集各含有 4922 和 3318 个样本实例。图像样本是通过 Amazon MechanicalTurk³⁷ 网站发起任务并有少量的操作人员参照指定规范完成的。该数据集对人物的头部进行标注是为了避免目标外包矩形任意比例的变化,避免引起标注歧义。标注的头部目标区域是以人眼可辨识为准则进行标注的,即使目标遮挡相当严重。收集的图像如图 6-5、图 6-6 所示。Brainwash 数据集对应的检测问题是一个含有小目标、部分遮挡、多尺度变换、衣着和容貌等挑战的图像检测问题。

实验部分还包括了 TUD-Crossing 数据集上的对比测试。该数据集图像记录的是

³⁶ <https://motchallenge.net/vis/TUD-Crossing>。

³⁷ Amazon Mechanical Turk 是一个互联网工作分发媒介平台。

人流量较的大街道，并已用于遮挡目标检测问题的算法验证。因为 TUD-Crossing 数据集不含有独立的训练集³⁸，实验是在 TUD-Brussels 数据集上进行训练的。原始的 TUD-Crossing 数据集真值标记中并不含有严重遮挡的行人目标。为了更好地评估不同方法在遮挡目标检测问题中的性能，实验对真值进行了扩展，使用了全部的行人标记。从原有的 1008 个行人目标增长至 1530 个行人目标。

目标检测成功的评判标准是检测位置与真值信息重叠比例大于 0.5。评估标准包括目标检测准确率、召回率、单张图像检测目标数量方差等。Brainwash 数据集和 TUD-Crossing 数据集它们都是存在目标遮挡的图像检测数据集。其中，TUD-Crossing 数据集的遮挡情况更加严重，如图 6-6 所示。

对比方法与性能评估

实验对比了 Faster-RCNN 算法和 OverFeat 算法。其中 Faster-RCNN 算法使用的是 VGGNet19 网络图像编码，而 OverFeat 算法与 ReInspect 算法都采用相同参数的 GoogLeNet 网络图像编码。



图 6-5 Brainwash 数据集实验检测结果示意图。上行图像表示的是 OverFeat 算法的检测结果。下行图像表示的是 ReInspect 算法的检测结果

38 TUD-Brussel 数据集与 TUD-Crossing 数据集相同的部分图像已在训练时移除。早前算法的对比实验可参阅本章参考资料[4, 18, 19]。

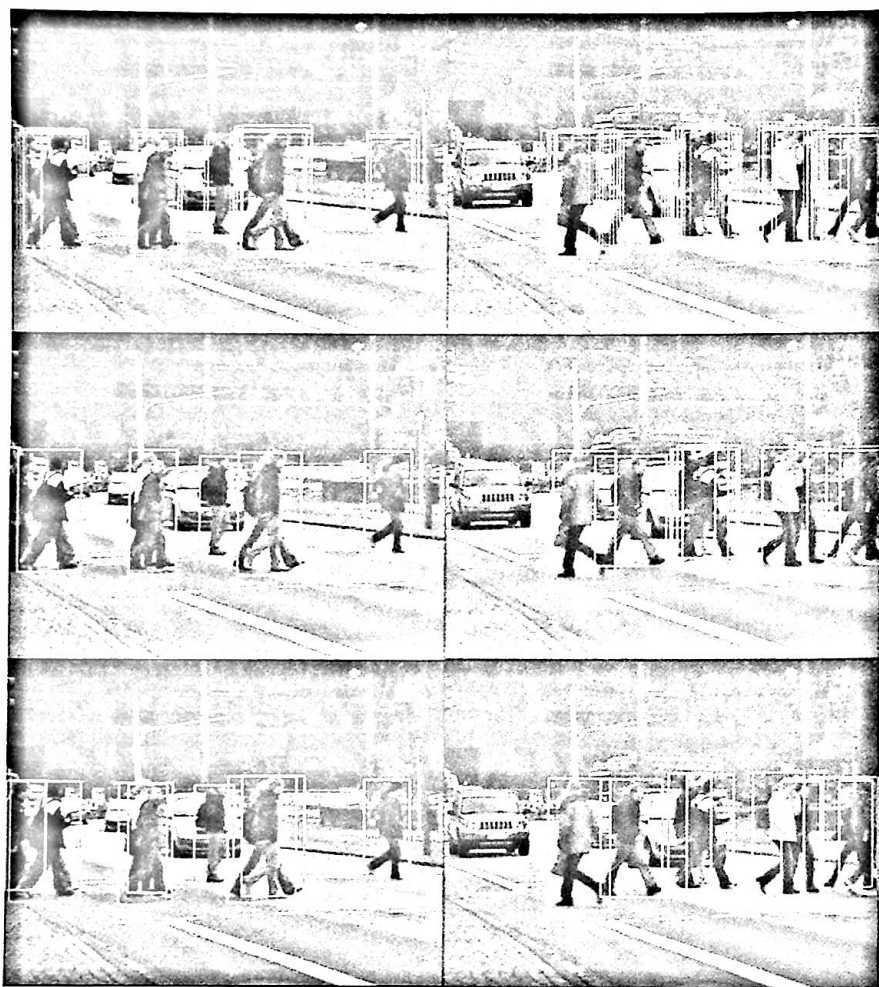


图 6-6 TUD-Crossing 数据集行人目标检测例图。中间一行和最下行的图像分别表示 $\tau = 0.75$ 时的 Faster R-CNN 算法和 ReInspect 算法的检测结果。ReInspect 算法的检测精度达到了 90%。最上行两张图像表示 Faster R-CNN 算法得出的所有未经过后处理的候选检测区域

对于 TUD-Crossing 数据集，相较于 OverFeat 算法，ReInspect 算法显现出了较大的性能提升，召回率从 71% 提升至 81%。另外，ReInspect 算法的整体精度也得到了提升。ReInspect 算法的整体精度为 0.78，而 OverFeat 算法的整体精度为 0.67。单张图像的人体计数误差分别是 0.76 和 1.05。Brainwash 数据集的实验结果同样呈现了类

似的结论, ReInspect 算法明显优于 OverFeat 算法。

图 6-5 显示了 OverFeat 算法和 ReInspect 算法在 Brainwash 数据集上的检测结果例图。ReInspect 算法能够检测出遮挡比较严重的头像, 而 OverFeat 算法则对此无能为力。因此, ReInspect 算法整体上要优于 OverFeat 算法。

图 6-6 显示了 Faster R-CNN 算法与 ReInspect 算法的检测示例结果。Faster R-CNN 算法后处理操作的重叠比例阈值参数($\tau \in [0,1]$)对算法的性能评估有非常大的影响。仅当候选检测结果重叠面积比例大于 τ 时会执行移除冗余重复检测的操作。实验部分包括(0.25 0.5 0.75)三个不同的阈值参数设置。在 TUD-Crossing 数据集中 ReInspect 算法的性能表现要好于所有参数设置的 Faster R-CNN 算法。对于 TUD-Crossing 数据集, 当 $\tau = 0.75$ 时, Faster R-CNN 算法可能会对同一目标产生冗余的检测结果。当 $\tau = 0.25$ 设置较小时, 冗余检测会被移除, 但会造成较多的遮挡目标漏检测。

Brainwash 数据集大部分目标的遮挡程度都小于 TUD-Crossing 数据集。Brainwash 数据集的测试结果表明, 仅当 $\tau = 0.5$ 参数设置的 Faster R-CNN 算法性能略优于 ReInspect 算法。因为 Faster R-CNN 算法使用了多个基于锚点的子窗口增强了目标检测召回率, 而 ReInspect 算法仅对单个网格特征进行分类预测计算。使用插值网格缩放特征后, ReInspect 算法能够使网格特征描述能力更强, 从而进一步提升检测性能。此时, ReInspect 算法性能将整体优于 Faster R-CNN 算法。在上一节的“GoogLeNet 网格图像编码”内容中对缩放特征有简单的介绍, 这里不再复述³⁹。

对于尺寸为 320x240 大小的三通道图像, TensorBox 中实现的 ReInspect 算法在较好的 GPU 设备上运行速率达到了 15 张/秒。TensorBox 为图像检测提供了一套完整的解决方案, 稍作修改该系统亦可用于图像分类与图像识别。

6.2 CNN+LSTM 网络模型与图像摘要问题

让计算机用文字来描述一张图像的内容, 是计算机视觉与自然语言处理的一项综合问题。使用深度学习技术中的 CNN 神经网络和 LSTM 循环神经网络将计算机视觉

39 关于缩放特征的详细描述可查阅本章参考资料[7]。

技术与机器翻译技术相结合就可以实现图像摘要系统。图像摘要系统的优化问题也就是图像内容至目标文字内容描述的映射转换问题。论文 *Show and Tell: Lessons learned from the 2015 MSCOCO Image Captioning Challenge* 实现了一种基于 TensorFlow 的图像摘要系统，称为 NIC (Neural Image Caption) 算法。该算法曾在 2015 年举行的 MSCOCO 图像摘要竞赛⁴⁰中取得了最好成绩，不仅在对图像内容判断的准确性方面，还在文字描述的流畅性方面，都表现出了不亚于人类的图像描述能力。

6.2.1 图像摘要问题

图像摘要算法是让计算机自动对图像产生文字描述的程序。描述的文字可以是英文、俄文、中文等，这只取决于程序的训练数据。问题的主要研究内容包括图像识别和自然语言处理两个方面，一方面需要实现对图像内容的准确提取，另一方面还需要实现对图像内容的恰当文字转换。其中，图像内容的准确提取包括图像目标的类型特征、场景类型特征、颜色特征、位置特征、大小特征等，而图像内容的文字转换则需要自动映射图像内容特征与文字特征的关系。

早前的图像摘要算法往往包括多个集成模块，例如图像目标检测、目标外观属性提取、位置信息提取、关联分析、候选单词预测、描述语句生成等。这些模块都对图像摘要的描述内容起到了关键作用。对于这些方法，图像描述的内容对象只可能是图像检测模块的输出结果，而对物体外观、位置等信息的描述，则都需要根据人工设定的语言逻辑先验信息来得到，这限制了描述语言的丰富程度。而 Oriol Vinyals 等提出的 NIC 算法使用深度学习技术一并解决了这些问题。该方法使用 inception_v3 网络模型统一提取图像高维特征，然后，使用 LSTM 网络结构实现对图像描述语句的优化。

简单来说，图像摘要问题的求解过程可以理解作为一种概率联合分布的转换模型。对于图像 I 和其对应的描述文本 $[S_{begin}, S_1, \dots, S_i, \dots, S_{end}]$ ，训练的目标为图像摘要的最大概率密度函数 $P(S_{begin}, S_1, \dots, S_{end} | I)$ 。在深度学习领域，对于序列联合概率密度优化问题早已在机器翻译中使用。对于原语言序列表示 $[S_{begin}, S_1, \dots, S_i, \dots, S_{end}]$ 和目标语言序列表示 $[T_{begin}, T_1, \dots, T_i, \dots, T_{end}]$ ，机器翻译的训练任务就是使联合概率模型的 $P(T_{begin}, T_1, \dots, T_i, \dots, T_{end} | S_{begin}, S_1, \dots, S_i, \dots, S_{end})$ 达到最大化。因此，与早期图像摘要算法类似，早前的机器翻译算法同样由多个模块构成（例如包括：单个单词翻译转换、

40 <http://mscoco.org/dataset/#captions-challenge2015>。

单词位置对齐、语言组织排序等)。而近年来循环神经网络在机器翻译方面的研究和应用已表明,使用循环网络能够更好地解决该问题。尤其以 Encoder-Decoder 结构的网络效果最为优秀。

借鉴机器翻译算法采用的思路,NIC 算法使用深度图像卷积网络 inception_v3 作为图像内容的编码器,同时使用 LSTM 网络作为对图像编码内容生成文字描述的解码器,这就为图像摘要问题提供了一种整体的解决方案,并通过 MSCOCO 竞赛结果验证了该方法的可靠性。

6.2.2 NIC 图像摘要生成算法

NIC 算法为图像摘要系统提供了一个基于深度学习的统一框架解决方案。首先,该方法将图像摘要问题抽象化为一个数字优化问题,如下所示:

$$\theta^* = \operatorname{argmax}_{\theta} \sum_{(I,S)} \log P(S|I, \theta)$$

其中, θ 表示深度神经网络的模型参数, I 表示图像数据, S 表示由单词序列构成的理想文字描述语句。考虑到 S 可以表示为任意的描述语句,对 S 表示的语句长度没有限制。因此,可以将该问题进一步转换为使用链式法则求解单个单词的最大联合概率密度。

$$\log P(S|I, \theta) = \frac{1}{N} \sum_{t=0}^N \log P(S_t|I, \theta, S_0, \dots, S_{t-1})$$

在训练时, (S, I) 构成训练样本的基本单元。整个训练过程就是上述公式最大值的优化过程。联合概率分布 $\log P(S_t|I, \theta, S_0, \dots, S_{t-1})$ 可以很容易地借助循环神经网络隐含状态层 h_t 模拟实现。在网络模型内部隐含状态层 h_t 由输入数据 x_{t-1} 、上一时刻状态层 h_{t-1} 共同决定,如下公式所示:

$$h_t = f(h_{t-1}, x_t)$$

使用循环网络衔接了模型优化的两个关键问题,一个是自定义非线性隐层转换函数 $f(h_{t-1}, x_t)$,另一个是可将图像信息和文字信息以序列形式同时组合形成输入

数据 x_t 。NIC 算法使用 LSTM 循环网络作为隐层的变换网络。输入数据由原始图像的特征和描述文本的词向量组合而成。其中图像特征由 inception_v3 网络处理得到, 文本词向量则要通过单词向量化方法获得。单词向量化是一种无监督学习, 其目标是将每一个单词都转换为向量表示, 并且语义越接近的单词的向量距离越近。

由上一章对循环神经网络的介绍可知, 普通 RNN 在序列数据的级联优化问题时, 会涉及反向传播梯度消失和梯度爆炸的关键问题, 而 LSTM 结构是解决这一问题的良药。因此, NIC 算法中使用 LSTM 循环网络作为摘要文本的输出模型, 如图 6-7 所示。

由公式 $\log P(S_t|I, \theta, S_0, \dots, S_{t-1})$ 可知, 在输入图像和单词序列后 LSTM 网络即可实现模型的训练或预测。动态 LSTM 循环网络很好地利用了 LSTM 网络的特性。即通过记忆单元的存储、更新策略, 最大化拟合图像内容和描述文字的关联性, 以及单词之间的语法规则。将循环网络展开, 可以理解为对 LSTM 网络的动态循环使用。图 6-7 中的 LSTM 网络都使用相同的模型参数。具体来说, 令图像数据表示为 I , 对应的描述语句单词序列表示为 (S_0, \dots, S_N) , 则预测单词按照如下公式进行:

$$x_{-1} = \text{CNN}(I)$$

$$x_t = W_e S_t, t \in \{0, 1, \dots, N-1\}$$

$$P_{t+1} = \text{LSTM}(x_t), t \in \{0, 1, \dots, N-1\}$$

其中, CNN 泛指图像数据的深度卷积网络, NIC 算法使用 inception_v3 网络。图像特征 x_{-1} 作为-1时刻的 LSTM 网络数据输入。如前文所说, S_0 和 S_N 分别表示描述语句的开始和结束标记。0时刻的数据输入为语句开始标记 $W_e S_0$ 特征。 S_t 表示由 $\{0, 1\}$ 元素组成的类型标签向量。 S_t 的长度为单词集合的总是 $|W|$, 且仅有一个非零元素。 W_e 表示尺寸为 $[|W|, Q]$ 的二维矩阵, 其中 Q 表示单词向量的长度。图 6-7 中略去了初始时的 LSTM 全零状态向量和最后的描述结束符标记 S_N 。图像特征和单词向量特征都映射到相同的特征空间中。图像原数据仅在网络中的 $t = -1$ 时输入 1 次作为 LSTM 网络最开始的输入数据。实践表明, 在图 6-7 中不同的时间节点多次导入图像特征向量只会降低算法的整体预测精度。这容易使网络产生更多的图像特征噪声且容易使模

型过拟合。整体模型的优化参数由 LSTM 网络参数、图像深度神经网络特征参数和单词向量化参数共同组成。

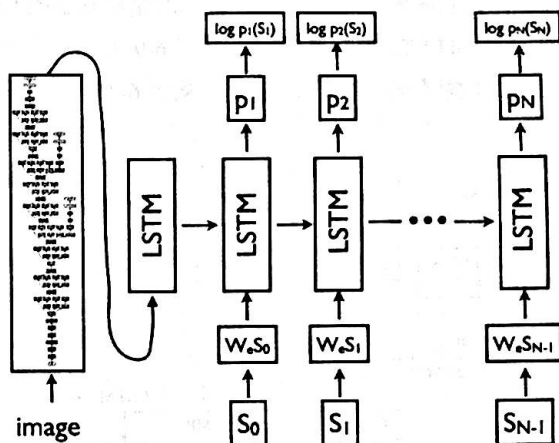


图 6-7 NIC 图像摘要算法流程图。LSTM 循环网络将图像向量化特征与单词向量化特征联系在一起。图中为 LSTM 循环网络的展开形式。蓝色箭头表示循环网络的数据流向。需要注意的是，图中所有标记的 LSTM 网络使用的都是相同的参数。图中略去了初始时的 LSTM 全零状态向量和最后的描述结束符标记 S_N 。 t 时刻的 LSTM 网络状态为 $t-1$ 时刻 LSTM 网络的输出

$$\log P(S|I, \theta) = \frac{1}{N} \sum_{t=0}^N \log P(S_t|I, \theta, S_0, \dots, S_{t-1})$$

TensorFlow 内部实现了多种不同类型的 LSTM 循环网络结构。在上一章对这部分内容有较详细的描述。为了不引起混淆，这里补充说明 LSTM 网络循环策略的多种不同形式。如图 6-8 所示，枚举了 4 种不同的 LSTM 网络循环方式。其中，图 6-8(a) 表示 LSTM 网络的自循环神经网络结构。该结构也是 NIC 算法采用的循环形式。 $t-1$ 时刻的 LSTM 网络输出将再次作为 t 时刻的输入进入 LSTM 网络。对于每一个输入序列，LSTM 网络将自动更新内部记忆单元。该结构的特点是参数较少，整个循环体仅有一个 LSTM 网络参数且对输入数据的变换仅通过 LSTM 网络内部的记忆单元和模型参数控制。图 6-8(b) 表示多个串联的 LSTM 网络的自循环神经网络。与图 6-8(a) 不同，该结构的循环网络包含多个独立的 LSTM 单元网络。这些单元内部的 LSTM

网络参数仅在循环内部共享。这种循环模式的网络结构参数更多训练更复杂，能够表征更复杂的模型函数，需要的样本也更多。图 6-8(c)表示基于 LSTM 的 RNN 循环网络。其结构示意图如图 6-9 右图所示。图 6-9 左图显示的是最基础的 LSTM 网络实现，对应于 TensorFlow 中 BasicLSTMCell 类的实现。图 6-9 左图与右图相比较的主要区别在于对输出层维数的控制以及网络参数的差异。从表 6-3 中可以看出这两种网络的计算差异。

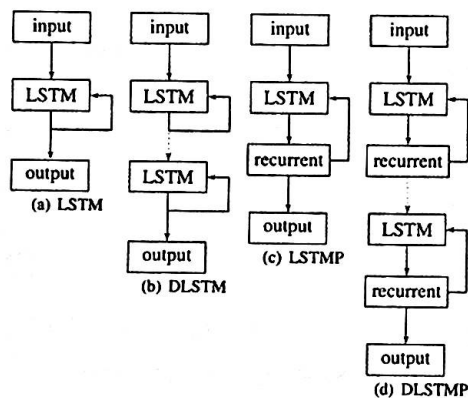


图 6-8 LSTM 网络循环策略示意图

表 6-3 标准 LSTM 网络与 LSTM RNN 网络计算公式

LSTM	$i_t = \text{sigm}(W_{ix}x_t + W_{im}m_{t-1} + b_i)$ $f_t = \text{sigm}(W_{fx}x_t + W_{fm}m_{t-1} + b_f)$ $o_t = \text{sigm}(W_{ox}x_t + W_{om}m_{t-1} + b_o)$ $c_t = f_t \odot c_{t-1} + i_t \odot h(W_{cx}x_t + W_{cm}m_{t-1})$ $m_t = o_t \odot c_t$ $p_{t+1} = \text{Softmax}(m_t)$
LSTM RNN	$i_t = \text{sigm}(W_{ix}x_t + W_{ir}r_{t-1} + b_i)$ $f_t = \text{sigm}(W_{fx}x_t + W_{fr}r_{t-1} + b_f)$ $o_t = \text{sigm}(W_{ox}x_t + W_{or}r_{t-1} + b_o)$ $c_t = f_t \odot c_{t-1} + i_t \odot h(W_{cx}x_t + W_{cr}r_{t-1})$ $m_t = o_t \odot c_t$ $r_t = W_{rm}m_t$ $p_{t+1} = \text{Softmax}(W_{ym}r_t)$

表 6-3 中的 LSTM 网络计算公式在上一章中已经说明。这里补充说明 LSTM RNN 网络的计算公式。该网络结构是在 LSTM 网络基础上新增了一个全连接层,并将连接结果重新导入到 LSTM 网络中。这一操作令网络的模型参数数量发生了变化。忽略全连接层的偏移量,标准 LSTM 网络的参数为 $N = N_c \times N_c \times 4 + N_i \times N_c \times 4 + N_o \times N_c + N_c \times 3$,而 LSTM RNN 网络的参数为 $N = N_c \times N_r \times 4 + N_r \times N_c \times 4 + N_o \times N_r + N_c \times N_r + N_c \times 3$ 。由此可知,在模型参数总数量不变的情况下,LSTM RNN 网络可设置 $N_r < N_c$ 增加记忆单元的维数,实践表明在参数数量相同的情况下 LSTM RNN 网络的性能整体要优于 LSTM 网络⁴¹。

LSTM RNN 神经网络是关于 LSTM 网络的改进版本。在 TensorFlow 内部可用函数 `raw_rnn` 实现。NIC 算法没有使用 LSTM RNN 神经网络,而是使用标准的 LSTM 神经网络。它的自循环方式为如图 6-8(a)所示的标准 LSTM 循环模式。

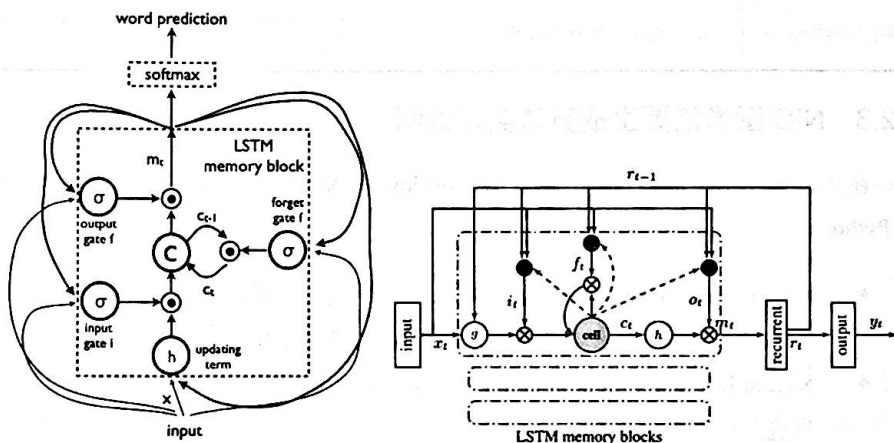


图 6-9 左图为 LSTM 循环网络结构示意图。右图为 LSTM RNN 循环网络结构示意图,其中网络的循环连接部分由蓝色标出。 $t-1$ 时刻的 LSTM 网络的输出将作用于 t 时刻的 LSTM 网络的输入。训练时,网络的循环次数由外部调用函数根据输入语句序列的长度控制。预测时,网络的循环次数由训练时的最大循环次数和终止符号预测结果共同决定

使用 NIC 算法有两种不同的单词序列生成方式。第一种称为采样方式,可以根据 $P(S_t|I, \theta, S_0, \dots, S_{t-1})$ 的概率采样生成 t 时刻的单词,而后再根据 $P(S_{t+1}|I, \theta, S_0, \dots, S_t)$

41 参见本章参考资料[27]。

的概率采样生成 $t+1$ 时刻的单词。直至生成结束符标记 S_N 或者达到语句的最大长度。另一种是优化搜索方法，在 t 时刻考虑该时刻联合置信度最大的 k 种单词序列生成结果，基于这些结果再往后预测 $t+1$ 时刻的置信度最大的 k 种单词序列生成结果。NIC 算法内部实现如表 6-4 所示。

表 6-4 NIC 算法内部实现

	训练阶段	预测阶段
数据输入及准备	使用 TFRecordReader 读取的训练数据	使用 Tensor 占位符
图像向量化	使用 InceptionV3 网络特征，创建全连接层抽取特征	读取 InceptionV3 网络特征和全连接层抽取特征
单词向量化	创建方差较小的随机二维矩阵	从模型文件中读取训练结果
定义模型模型结构操作	创建基于 LSTM 的动态循环网络	创建单个 LSTM 网络依次预测单个序列单词
神经网络的运行	运行损失函数优化操作	运行单词预测操作并搜索返回最优结果

6.2.3 NIC 图像摘要生成算法实现说明

在开始 NIC 算法的调试前，除了 TensorFlow 和 NumPy，需要安装下列相关软件包和 Python 包。

- Bazel：它是 Google 的一款可再生的代码构建工具。在本例中负责完成对 Python 脚本的程序化封装。使脚本能够按照多进程方式运行。
- Natural Language Toolkit (NLTK)：它是一个基于 python 语言的自然语言处理工具包。

本例中并没有使用 Bazel 和 NLTK 太多复杂功能。使用 md 文件阅读器打开 README.md 文件可知本例中如何使用 Bazel 对 Python 脚本进行封装。从编辑结果可知，Bazel 生成的 Python 编译结果实际上只是对 Python 脚本程序的进程调用。在 Ubuntu 系统下，进入项目主目录，输入以下脚本指令即可实现对数据预处理、训练和预测脚本的进程封装。关于 Bazel 的封装选项可查看对应目录下的 build 文件。详细用法请查阅 Bazel 教程。新生成的封装结果都将在 `bazel-bin/im2txt` 目录下。

```
bazel build im2txt/downloadandpreprocess_mscoco
bazel build -c opt im2txt/...
```

```
bazel build -c opt im2txt/run_inference
```

在 NIC 算法的 TensorFlow 实现版本中用到了 NLTK(Natural Language Toolkit)自然语言 Python 工具包。使用该工具包的主要目的是实现对样本描述语句单词与标点符号的自动拆分。如下代码所示, `caption` 是原始图像数据的描述语句字符串, 且 `caption` 由单词与标点符号组成。

```
import nltk.tokenize
...
def _process_caption(caption):
    """Processes a caption string into a list of tokenized words.

    Args:
        caption: A string caption.

    Returns:
        A list of strings; the tokenized caption.
    """
    tokenized_caption = [FLAGS.start_word]

    tokenized_caption.extend(nltk.tokenize.word_tokenize(caption.lower(
)))
    tokenized_caption.append(FLAGS.end_word)
    return tokenized_caption
```

由此可见,无论是 Bazel 或是 NLTK 工具包对 NIC 算法的实现都没有起到太大作用。因此,本节后续内容将不再对这两个依赖部分进行说明。实际上从算法原理上来看, NIC 算法中最复杂的部分是 LSTM 网络的构建。但从实际的程序实现和运行效果上来看,最开始的数据处理程序比模型构建程序要复杂得多且对最终的训练效果同样起到非常重要的影响。若没有前期的数据处理程序为深度神经网络快速提供大量多样化的数据,整个网络性能将无法评估。因此有必要对网络的数据输入程序部分引起足够的重视。对于大数据的训练问题更是如此。

输入数据及准备

使用 Bazel 封装好 NIC 算法的各模块后,还需要准备训练数据。在第 3 章曾经提到过, TensorFlow 中有三种不同的训练数据输入方式。一种是定义占位符,在运行时输入具体数据的方式。这种方式对数据处理比较灵活,适合数据量适中 I/O 读写不太

频繁的情况。第二种是使用预选加载的数据方式。这种方式数据加载方式简单，数据输入几乎没有 I/O 操作。适合用于测试网络结构。第三种是使用训练文件，这种方式专门针对大数据的训练，训练数据应使用 TensorFlow 指定格式保存。TensorFlow 内部已对这种情况下的 I/O 操作进行了优化。

大数据的学习训练是深度学习的重要特性之一。利用数据包的训练机制，深度学习算法能够训练上百、上千 GB 的数据。然而，对硬盘数据的频繁的读取/写入操作对训练算法提出了新的问题。对于大数据上百万次的读取/写入操作还需要有专门的算法确保数据读取的有效性和速度。对此，TensorFlow 提供了专有的训练数据读取/写入接口，`tensorflow.TFRecordReader()`和 `tensorflow.TFRecordWriter()`。NIC 算法同时使用了这两个函数。在本例中它们都是使用 `tf.train.SequenceExample` 对象实现数据的读取和写入的。而 `tf.train.SequenceExample` 对象保存的是变量的序列化结果。整个数据处理过程如表 6-5 所示。因模型的训练需要先定义结构后再放入到 Session 中运行，因此数据输入端同样要先定义为模型结构图的组成部分，而后再在 Session 中运行。这是通过 TensorFlow 的队列机制实现的。函数 `tf.train.queue_runner.QueueRunner` 负责创建队列操作。函数 `tf.train.queue_runner.add_queue_runner` 则将参数队列投入到指定的 Session 中运行。本节后续内容将以 NIC 算法为例对基于 `tensorflow.TFRecord` 格式的大数据使用方式进行详细说明。在预测时，NIC 算法能够对单张输入图像返回描述语句。此时，网络的数据输入只包含图像数据，且是按占位符的方式传入的。这种数据传入方式前文已经描述了很多，这里不再复述。

表 6-5 NIC 算法中基于 TFRecord 格式的大数据准备与使用过程

转换数据写入 TFRecord 过程	训练时 TFRecord 数据使用过程
1. 读取全部原始数据的摘要信息。包括路径、字符串描述等	1. 使用 <code>tf.gfile.Glob</code> 函数载入 TFRecord 文件文件名列表
2. 根据训练问题和原始数据集的大小，重新调整训练集、验证集和测试集的比例大小	2. 使用 <code>tf.train.string_input_producer</code> 函数创建文件名字符串无限循环队列。在训练过程中，将自动加载队列中的文件输入样本
3. 根据写入数据的线程数，将数据集（训练集、验证集和测试集执行相同操作）的样本数量等分为相同大小的数据块。每一个数据块由单独的线程负责写入数据	3. 根据线程参数， <code>TensorFlow.TFRecordReader</code> 对象读取文件名字符串队列中的文件。使用 <code>tf.train.queue_runner.QueueRunner</code> 函数定义操作，将读取的数据文件内容添加到保存数据的列表对象中。再使用 <code>tf.train.queue_runner.add_queue_runner</code> 函数将其

续表

转换数据写入 TFRecord 过程	训练时 TFRecord 数据使用过程
	添加到当前会话中运行，以获取 <code>tf.train.SequenceExample</code> 序列化数据
4. 根据输入参数，线程内部再对数据块细分为固定大小的分块，并准备使用 <code>TensorFlow.TFRecordWriter</code> 对象将单个数据样本依次写入 TFRecord 文件	4. 定义操作解析 <code>tf.train.SequenceExample</code> 序列化数据，并对单个样本数据中的图像内容部分进行随机调整强化训练数据的多样性
5. 分别使用 <code>tf.train.FeatureList</code> 和 <code>tf.train.Feature</code> 函数对样本数据进行序列化操作。最后使用 <code>tf.train.SequenceExample</code> 构建序列化样本写入文件	5. 跟单个样本的描述语句生成序列化标签。使用 <code>tf.train.batch_join</code> 函数定义操作将单个样本文件组合构建训练用的 batch 数据包传入训练网络

原始的 MSCOCO 图像摘要竞赛数据集为 20GB 大小的数据信息压缩包。每张图像含有 5 种不同的描述方式。为了最大化训练数据的输入速度并确保数据读取的有效性，训练程序使用了 `tensorflow.TFRecordReader()` 和线程队列的方式实现数据的快速输入。因此，训练数据需要使用对应的 `tensorflow.TFRecordWriter()` 函数写入。考虑到这种数据的写入方式要求图像数据和描述语句一一对应，故原始 MSCOCO 数据解压后每张图像数据按照其描述语句的次数被重复写入了多次。处理完毕后，整体训练集的大小为 120GB 左右。

另外，可从项目文件 `im2txt\data\download_and_preprocess_mscoco.sh` 中了解到原始 MSCOCO 数据的下载地址。脚本程序使用 `wget` 命令下载。若下载失败，可改用 `uget` 等专用下载工具下载获取。可执行如下语句自动转换生成训练数据集。

```
bazel-bin/im2txt/downloadandpreprocessmscoco "${MSCOCODIR}"
```

NIC 算法的模型参数由 LSTM 网络参数、图像深度神经网络特征参数和单词向量化参数共同组成。其中，单词向量化参数的总数占总体参数的大部分比例。因此，单词向量化在图像摘要优化问题中是一个关键的优化子问题。考虑到 MSCOCO 数据集单词数量为 29415 个，而且在图像描述语句中出现的频率差异较大。因此有必要筛选掉一些在描述语句中出现频率较小的单词。在 NIC 的默认算法参数将移除出现频率小于 4 的生僻单词并将其全部替换为“nul”表示。因此，NIC 算法中的描述词汇量大小为 11519 个不同的单词。

生成数据脚本实际上的执行的是 `data/build_mscoco_data.py` 文件中的指令。从这个文件中可以发现 TensorFlow 的一些 I/O 写入操作并不需要定义完毕后再在 Session

运行。程序使用 `_load_and_process_metadata` 函数读取原始 MSCOCO 数据集。输入的两个参数分别表示图像摘要真值文件和对应的图像文件存放文件夹路径。原始 MSCOCO 数据训练集和验证集分别含有 82783 和 40504 张图像。因公开的 MSCOCO 数据没有测试数据集,故算法最后将训练集和验证集重新根据比例组合形成新的训练集、验证集和测试集。新的训练集、验证集和测试集分别含有 117211、2025 和 4051 张图像。使用这样的数据划分是为了确保足够多的样本参与训练。训练集、测试集、验证集的重新分配代码如下所示。

```
# Load image metadata from caption files.
mscoco_train_dataset = _load_and_process_metadata(FLAGS.train_
captions_file,
                                                    FLAGS.train_image_
dir)
mscoco_val_dataset = _load_and_process_metadata(FLAGS.val_captions_
file,
                                                    FLAGS.val_image_dir)

# Redistribute the MSCOCO data as follows:
#   train_dataset = 100% of mscoco_train_dataset + 85% of
mscoco_val_dataset.
#   val_dataset = 5% of mscoco_val_dataset (for validation during
training).
#   test_dataset = 10% of mscoco_val_dataset (for final
evaluation).
train_cutoff = int(0.85 * len(mscoco_val_dataset))
val_cutoff = int(0.90 * len(mscoco_val_dataset))
train_dataset = mscoco_train_dataset + mscoco_val_dataset
[0:train_cutoff]
val_dataset = mscoco_val_dataset[train_cutoff:val_cutoff]
test_dataset = mscoco_val_dataset[val_cutoff:]
```

从单词向量化的参数来看,令词典共有 11519 个单词需要用 512 维的向量表示,则共需要优化 $11519 \times 512 = 5897728$ 个参数。从参数和样本的数量上可以看出 $5897728 : 117211 \approx 50 : 1$, 一张图像摘要信息样本对应 50 个参数。考虑到描述语句的平均长度为 13 个单词,因此一个单词对应的优化参数数量平均大约在 5 个左右。由此可见,模型参数的因变量区间范围大致上能够反映训练样本集的数据信息并存在一定的冗余空间。

训练集的描述语句按照单词为基本单元进行了拆分。最终以单词为基本单元构成集合 `vocab`。列表 `train_captions` 中保存的是图像描述语句字符串列表。函数 `_create_vocab` 实现对字符串的单词分解并转换为类的对象。`vocab` 是含有训练集中出现过的所有单词的字典类型。它含有两个成员变量。其中, `_vocab` 被用于保存单词到序号的映射字典, 而 `_unk_id` 则保存对应字典的总长度。`_vocab` 中的所有单词都按照词频从大到小做排序。

```
# Create vocabulary from the training captions.
train_captions = [c for image in train_dataset for c in
image.captions]
vocab = _create_vocab(train_captions)
```

最后函数 `_process_dataset` 内部会按照多线程方式将输入数据以指定的名字参数作为后缀存入硬盘。`_process_dataset` 有 4 个输入参数。它们的作用分别是存入数据文件名、原始数据路径信息、单词信息和写入数据的分块文件数量。其中, 原始数据路径信息是指图像文件的 id 标示号、图像文件名路径、单张图像的所有摘要内容构成的三元组。写入数据的分块文件数量还必须大于线程数量, 且能够被线程数量整除。因原始数据单张图像对应多个摘要描述, 需要对数据进行转换, 以确保存入的图像和摘要一一对应再保存数据。

```
_process_dataset("train", train_dataset, vocab, FLAGS.train_shards)
_process_dataset("val", val_dataset, vocab, FLAGS.val_shards)
_process_dataset("test", test_dataset, vocab, FLAGS.test_shards)
```

`_process_dataset` 函数内部关键代码如下所示。首先使用两个嵌套 `for` 语句将原始样本中单个图像对应多个摘要信息转换为一对一关系并重新保存为列表。变量 `caption` 是字符串描述语句。这里将其保存为列表结构是方便后续处理的单词列表转换。

```
def _process_dataset(name, images, vocab, num_shards):
    .....
    images = [ImageMetadata(image.image_id, image.filename, [caption])
              for image in images for caption in image.captions]
```

至此, `images` 保存了写入数据的全部路径信息。为了提升输入写入速度, `images` 信息将根据线程数量拆分为等长的数据块, 每个数据分块由单个线程保存为多个文件。

```

num_threads = min(num_shards, FLAGS.num_threads)
spacing = np.linspace(0, len(images), num_threads +
1).astype(np.int)
.....
for i in xrange(len(spacing) - 1):
    ranges.append([spacing[i], spacing[i + 1]])

```

最后下列语句开启相互无关多线程操作,将分块的数据依次保存为多个数据文件。

```

coord = tf.train.Coordinator()
decoder = ImageDecoder()
for thread_index in xrange(len(ranges)):
    args = (thread_index, ranges, name, images, decoder, vocab,
num_shards)
    t = threading.Thread(target=_process_image_files, args=args)
    t.start()
    threads.append(t)
coord.join(threads)

```

写入数据线程函数 `_process_image_files` 的关键代码如下所示。参数 `num_shards` 用以设置整个数据集将拆分为文件的数量。每个线程将产生 `int(num_shards / num_threads)` 个文件,该值由 `num_shards_per_batch` 表示。`shard_ranges` 表示线程内部的数据分块标记。`num_images_in_thread` 表示线程负责写入的数据样本总数。

```

def _process_image_files(thread_index, ranges, name, images,
decoder, vocab, num_shards):
    num_threads = len(ranges)
    num_shards_per_batch = int(num_shards / num_threads)
    shard_ranges = np.linspace(ranges[thread_index][0], ranges
[thread_index][1], num_shards_per_batch + 1).astype(int)
    num_images_in_thread = ranges[thread_index][1] - ranges[thread_
index][0]

```

线程函数内部使用两个循环操作来实现文件的写入操作。外层循环创建一个 `tf.python_io.TFRecordWriter` 写入文件对象。随后, `writer.write(sequence_example.SerializeToString())` 指令负责将图像和摘要信息写入文件。`writer.close()` 关闭写文件对象。其中, `output_filename` 变量用于保存当前的写入文件名,例如“train-00002-of-000256”,则此时 `name` 为“train”,`shard` 值为 2,`num_shards` 值为 256。

```

counter = 0
for s in xrange(num_shards_per_batch):

```

```

# Generate a sharded version of the file name, e.g.
'train-00002-of-00010'
shard = thread_index * num_shards_per_batch + s
output_filename = "%s-%.5d-of-%.5d" % (name, shard, num_shards)
output_file = os.path.join(FLAGS.output_dir, output_filename)
writer = tf.python_io.TFRecordWriter(output_file)
shard_counter = 0
images_in_shard = np.arange(shard_ranges[s], shard_ranges[s + 1],
dtype=int)

```

内层循环将依次读取样本信息然后写入文件。函数 `_to_sequence_example` 根据输入训练样本信息读取硬盘文件对应的编码图像数据, 返回 `tf.train.SequenceExample` 对象。若指定的文件解码失败, 则跳过该文件。

```

for i in images_in_shard:
    image = images[i]
    sequence_example = _to_sequence_example(image, decoder,
vocab)
    if sequence_example is not None:
        writer.write(sequence_example.SerializeToString())
        shard_counter += 1
        counter += 1
.....
writer.close()
.....

```

函数 `_to_sequence_example` 的关键代码如下所示。`tf.train.SequenceExample` 对象是 TensorFlow 的标准训练数据保存形式。函数 `tf.gfile.FastGFile` 负责快速打开并读取二进制文件内容, 并保存至 `encoded_image` 对象中。使用自定义类 `decoder.decode_jpeg` 尝试解码 jpg 图像。若解码失败则返回退出函数, 跳过该文件的信息写入操作。

```

def _to_sequence_example(image, decoder, vocab):
    with tf.gfile.FastGFile(image.filename, "r") as f:
        encoded_image = f.read()
    try:
        decoder.decode_jpeg(encoded_image)
    except (tf.errors.InvalidArgumentError, AssertionError):
        print("Skipping file with invalid JPEG data: %s" %
image.filename)
    return

```

TensorFlow 中实现了 jpeg 图像解码的操作。但该操作必须放在具体的 Session 会话中运行。因此 decoder 对象的对应类 ImageDecoder 中含有一个 tf.Session 对象。

```
class ImageDecoder(object):
    """Helper class for decoding images in tensorflow."""

    def __init__(self):
        # Create a single TensorFlow Session for all image decoding calls.
        self._sess = tf.Session()

        # TensorFlow ops for JPEG decoding.
        self._encoded_jpeg = tf.placeholder(dtype=tf.string)
        self._decode_jpeg = tf.image.decode_jpeg(self._encoded_jpeg,
                                                  channels=3)

    def decode_jpeg(self, encoded_jpeg):
        image = self._sess.run(self._decode_jpeg,
                                feed_dict={self._encoded_jpeg:
encoded_jpeg})
        assert len(image.shape) == 3
        assert image.shape[2] == 3
        return image
```

若文件为 jpg 文件，则按如下形式使用图像的 jpg 编码数据和图像 id 号构建 tf.train.Features 对象 context。

```
context = tf.train.Features(feature={
    "image/image_id": _int64_feature(image.image_id),
    "image/data": _bytes_feature(encoded_image),
})
```

另一方面，程序使用之前介绍的字典对象 vocab 将摘要语句转换为对应的序号序列。图像描述内容与对应的序号序列构建 tf.train.FeatureLists 对象 feature_lists。最终的训练样本数据由 context 和 feature_lists 构成 tf.train.SequenceExample 对象 sequence_example。

```
assert len(image.captions) == 1
caption = image.captions[0]
caption_ids = [vocab.word_to_id(word) for word in caption]
feature_lists = tf.train.FeatureLists(feature_list={
    "image/caption": _bytes_feature_list(caption),
```

```

        "image/caption_ids": _int64_feature_list(caption_ids)
    })
    sequence_example = tf.train.SequenceExample(
        context=context, feature_lists=feature_lists)

    return sequence_example

```

上述函数中调用的 `_int64_feature`、`_bytes_feature`、`_bytes_feature_list`、`_int64_feature_list` 等函数，作用都是将变量序列化转换。它们内部都是调用了 TensorFlow 的序列化接口。这些函数都要求输入参数为限定的类型。相关代码如下。

```

def _int64_feature(value):
    """Wrapper for inserting an int64 Feature into a SequenceExample
    proto."""
    return
    tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
def _bytes_feature(value):
    """Wrapper for inserting a bytes Feature into a SequenceExample
    proto."""
    return
    tf.train.Feature(bytes_list=tf.train.BytesList(value=[str(value)]))
def _int64_feature_list(values):
    """Wrapper for inserting an int64 FeatureList into a SequenceExample
    proto."""
    return tf.train.FeatureList(feature=[_int64_feature(v) for v in
    values])
def _bytes_feature_list(values):
    """Wrapper for inserting a bytes FeatureList into a SequenceExample
    proto."""
    return tf.train.FeatureList(feature=[_bytes_feature(v) for v in
    values])

```

上述代码实现了 MSCOCO 数据的转换，因为原始数据中一张图像对应了多个描述语句。原有的 20GB 图像数据被解压后转换为 120GB 的训练数据。转换后的数据更易于 TensorFlow 的快速读取。

完成数据的转换后即可加载转换数据文件输入到训练网络中。在上一节中提到了使用 `FIFOQueue` 队列与占位符的方式获取训练数据。这里将介绍使用 `RandomShuffleQueue` 队列与文件输入的方式来获取训练数据。TensorFlow 提供了 `tf.Coordinator` 和 `tf.QueueRunner` 这两个类与队列对象配合使用。需要说明的是上一节

中没有使用 `tf.Coordinator` 类，而是直接使用 Python 的线程机制实现队列的异步线程调用。另外，在本例中是直接使用 `TFRecord` 文件作为训练数据源。此时，除了占位符方式，TensorFlow 还为其他函数提供更简便的数据输入方式，能够将文件输入操作定义为网络结构的一部分。

针对 `TFRecord` 文件，函数 `tf.train.string_input_producer` 提供了简便的训练数据加载方式。该函数返回一个字符串队列并可用于构建 TensorFlow 的训练图，还将硬盘文件作为定义网络图的数据输入。该函数接口定义如表 6-6 所示。

表 6-6 `tf.train.string_input_producer` 函数接口定义

函数声明	<code>tf.train.string_input_producer(string_tensor, num_epochs=None, shuffle=True, seed=None, capacity=32, shared_name=None, name=None, cancel_op=None)</code>
函数功能	返回一个可作为训练输入管道的字符串队列
参数说明	
<code>string_tensor</code>	表示输出字符串队列的内容，是一个 1 维的字符串张量
<code>num_epochs</code>	可选参数。若该参数不为空则应是一个有效数值。当 <code>num_epochs</code> 为空时，返回一个元素由 <code>string_tensor</code> 组成的无限循环队列。否则，函数内部开启一个计数器，返回元素由 <code>string_tensor</code> 构成且最多循环次数为 <code>num_epochs</code> 的队列
<code>shuffle</code>	可选参数。若该值为真，在每次循环中随机打乱返回队列各元素的位置
<code>seed</code>	可选参数。整形数值参数，作为随机种子值，当 <code>shuffle</code> 为真时该值影响队列元素的乱序效果
<code>capacity</code>	可选参数。表示返回队列的长度，默认为 32
<code>shared_name</code>	可选参数。可在不同 Session 会话中的共享名称
<code>name</code>	可选参数，操作名称
<code>cancel_op</code>	可选参数，关于取消该队列的操作名称

NIC 算法在训练或评估时，数据的输入操作是由 `ops/inputs.py` 文件中的 `prefetch_input_data` 函数实现的。该函数首先使用文件名列函数 `tf.train.string_input_producer` 创建文件名字符串循环队列。作为传入参数，字符串张量 `data_files` 表示存放在硬盘的 `TFRecord` 文件路径列表。

NIC 算法源码中数据文件网络输入操作定义如以下代码所示。其中，`filename_queue` 被定义为随机排序的最大尺寸为 16 的字符串队列。`values_queue` 被定义为一个最小样本量为 4600（默认时 `values_per_shard` 为 2300，`input_queue`

capacity_factor 为 2), 最大容量 7800 的 RandomShuffleQueue 随机队列。values_queue 被用于保存从文件中读取的数据序列信息。

在验证时, filename_queue 则被定义为长度为 1 的字符串队列。values_queue 被定义为最大容量为 2396 (默认时 values_per_shard 为 2300, batch_size 为 32) 的 FIFOQueue 队列。

```
data_files = []
for pattern in file_pattern.split(","):
    data_files.extend(tf.gfile.Glob(pattern))
if is_training:
    filename_queue = tf.train.string_input_producer(
        data_files, shuffle=True, capacity=16, name=shard_queue_
name)
    min_queue_examples = values_per_shard * input_queue_capacity_
factor
    capacity = min_queue_examples + 100 * batch_size
    values_queue = tf.RandomShuffleQueue(
        capacity=capacity,
        min_after_dequeue=min_queue_examples,
        dtypes=[tf.string],
        name="random_" + value_queue_name)
else:
    filename_queue = tf.train.string_input_producer(
        data_files, shuffle=False, capacity=1, name=shard_queue_
name)
    capacity = values_per_shard + 3 * batch_size
    values_queue = tf.FIFOQueue(
        capacity=capacity, dtypes=[tf.string], name="fifo_" +
value_queue_name)
```

完成了文件名队列和数据队列的声明后, 还需要定义它们之间的关联操作。首先需要使用 TFRecordReader 对象 reader 定义读取文件操作, 然后定义读取内容的入队列操作。语句 enqueue_ops.append(values_queue.enqueue([value])) 定义了一次入队列操作, 并将该操作存入到列表对象 enqueue_ops。for 循环使 enqueue_ops 含有 num_reader_threads 个入队列操作。最后, tf.train.queue_runner.QueueRunner 实现了数据队列的多线程入队列定义。tf.train.queue_runner.add_queue_runner 将该操作加入到默认的 Session 会话中。

```

enqueue_ops = []
for _ in range(num_reader_threads):
    _, value = reader.read(filename_queue)
    enqueue_ops.append(values_queue.enqueue([value]))

tf.train.queue_runner.add_queue_runner(tf.train.queue_runner.QueueRunner(values_queue, enqueue_ops))
return values_queue

```

至此，构建了 TFRecord 训练数据的输入操作。ops/inputs.py 文件中的 prefetch_input_data 函数将返回数据队列 input_queue。该队列 input_queue 的出队列操作 dequeue() 将返回一个存入时的 tf.train.SequenceExample 对象序列化对象样本。因此，还需要定义 tf.train.SequenceExample 对象序列化对象样本的解析操作。

parse_sequence_example 函数同样在 ops/inputs.py 文件中。该函数输入三个参数，其中 serialized 为读取的序列化单个样本数据，image_feature 和 caption_feature 为数据标签。在 configuration.py 中可查读取数据时的标签变量 image_feature 和 caption_feature 的实际值分别为 “image/data” 和 “image/caption_ids”。这与存入时的标签名称是一致的。返回编码图像数据张量 encoded_image 和描述语句单词序号序列 caption。

```

def parse_sequence_example(serialized, image_feature,
caption_feature):
    """Parses a tensorflow.SequenceExample into an image and caption.

    Args:
        serialized: A scalar string Tensor; a single serialized
        SequenceExample.
        image_feature: Name of SequenceExample context feature
        containing image
        data.
        caption_feature: Name of SequenceExample feature list containing
        integer
        captions.

    Returns:
        encoded_image: A scalar string Tensor containing a JPEG encoded
        image.
        caption: A 1-D uint64 Tensor with dynamically specified length.
    """

```

```

context, sequence = tf.parse_single_sequence_example(
    serialized,
    context_features={
        image_feature: tf.FixedLenFeature([], dtype=tf.string)
    },
    sequence_features={
        caption_feature: tf.FixedLenSequenceFeature([],
dtype=tf.int64),
    })

```

```

encoded_image = context[image_feature]
caption = sequence[caption_feature]
return encoded_image, caption

```

到目前为止 TenosrFlow 还无法直接使用编码图像数据进行图像摘要的训练。还需要定义图像的解码操作。另外，为了扩充图像样本的多样性，有必要对解码的图像添加一些随机变化。这些操作都是在 ops/image_processing.py 文件中的 process_image 函数定义的。其中图像解码操作定义如以下代码所示，这里使用了 tf.image.decode_jpeg 函数和 tf.image.decode_png 函数定义解码操作，但并没有在 Session 中实际运行。

```

# Decode image into a float32 Tensor of shape [?, ?, 3] with values
in [0, 1).
with tf.name_scope("decode", values=[encoded_image]):
    if image_format == "jpeg":
        image = tf.image.decode_jpeg(encoded_image, channels=3)
    elif image_format == "png":
        image = tf.image.decode_png(encoded_image, channels=3)
    else:
        raise ValueError("Invalid image format: %s" % image_format)
    image = tf.image.convert_image_dtype(image, dtype=tf.float32)

```

在前文的样本和模型参数数量比较分析中提到了，一张图像摘要信息样本对应 50 个模型参数。考虑到描述语句的平均长度为 13 个单词，因此一个单词对应的优化参数数量平均大约在 5 个左右。模型参数的因变量区间范围大致上能够反映训练样本集的数据信息并存在一定的冗余空间。也就是说，这里使用了较多的参数与较少的样本，容易产生过拟合问题。训练完毕后，在不影响视觉内容的前提下，对训练集中的任一张图像略作修改，都可能会产生与原始训练图像差异较大的模型输出结果。为了增强并验证模型对这一现象的处理能力，有必要在训练过程中随机对图像数据进行修

改调整。这种随机修改调整包括图像尺寸、位置、色调、灰度、对比度等多个方面。

inception_v3 网络的训练输入图像尺寸为[299,299]。为了确保输入图像尺寸与 inception_v3 网络对应,该函数还定义了图像的缩放和裁剪操作。将图像统一缩放为 [346,346]大小,并随机截取在[299,299]大小的子图返回。以确保图像位置和尺寸上输入内容的多样性。

```
# Resize image.
assert (resize_height > 0) == (resize_width > 0)
if resize_height:
    image = tf.image.resize_images(image,
                                   size=[resize_height, resize_width],
                                   method=tf.image.ResizeMethod.BILINEAR)

# Crop to final dimensions.
if is_training:
    image = tf.random_crop(image, [height, width, 3])
else:
    # Central crop, assuming resize_height > height, resize_width >
width.
    image = tf.image.resize_image_with_crop_or_pad(image, height,
width)
```

另一方面, distort_image 函数扩充了图像内容的多样性。最后,减法操作和乘法操作确保图像数据在[-1, 1]的取值区间范围内。这是为了同 inception_v3 深度神经网络训练好的默认参数输入数据范围相对应。

```
# Randomly distort the image.
if is_training:
    image = distort_image(image, thread_id)

image_summary("final_image", image)

# Rescale to [-1,1] instead of [0, 1]
image = tf.sub(image, 0.5)
image = tf.mul(image, 2.0)
return image
```

distort_image 函数内部定义了多个随机图像内容修改操作。函数 tf.image.random_flip_left_right 将随机水平翻转图像内容。另外,该函数还将根据线程

序号的奇偶数随机修改图像内容的亮度、饱和度、色调和对比度等。这些变换操作是分别通过 `tf.image.random_brightness`、`tf.image.random_saturation`、`tf.image.random_hue`、`tf.image.random_contrast` 等内置函数实现的。`tf.clip_by_value` 函数确保修改后的图像内容取值在 $[0,1]$ 范围内，以确保是有效的图像数据。

```
# Randomly flip horizontally.
with tf.name_scope("flip_horizontal", values=[image]):
    image = tf.image.random_flip_left_right(image)

# Randomly distort the colors based on thread id.
color_ordering = thread_id % 2
with tf.name_scope("distort_color", values=[image]):
    if color_ordering == 0:
        image = tf.image.random_brightness(image, max_delta=32. /
255.)
        image = tf.image.random_saturation(image, lower=0.5,
upper=1.5)
        image = tf.image.random_hue(image, max_delta=0.032)
        image = tf.image.random_contrast(image, lower=0.5, upper=1.5)
    elif color_ordering == 1:
        image = tf.image.random_brightness(image, max_delta=32. /
255.)
        image = tf.image.random_contrast(image, lower=0.5, upper=1.5)
        image = tf.image.random_saturation(image, lower=0.5,
upper=1.5)
        image = tf.image.random_hue(image, max_delta=0.032)

# The random_* ops do not necessarily clamp.
image = tf.clip_by_value(image, 0.0, 1.0)
```

在上一节中介绍的图像检测问题样本数据是以图像为基本单位输入到网络中进行训练的。因为，图像内部会以网格方式划分为多个样本构建 `batch` 训练包进行训练。在图像摘要问题中，若以单个样本为基本单元反向梯度逐步修正整个训练集，则可能更易导致训练速度缓慢甚至梯度发散的情况。因此，有必要对输入数据构建 `batch` 包再传入训练网络。关于 `batch` 包的相关含义将在下一章中进行介绍。`ops/inputs.py` 文件中的 `batch_with_dynamic_pad` 函数定义了输入数据的 `batch` 包构建操作。

如前文所述，下列代码从 `TFRecord` 文件中读取了 `tf.train.SequenceExample` 序列化对象数据保存为 `serialized_sequence_example`。再通过 `input_ops.parse_sequence_`

example 函数定义了图像解码操作。接着, self.process_image 函数定义了图像内容的随机修改。外层 for 循环令 self.config.num_preprocess_threads(默认值为 4)个样本组成列表对象 images_and_captions。

```
images_and_captions = []
for thread_id in range(self.config.num_preprocess_threads):
    serialized_sequence_example = input_queue.dequeue()
    encoded_image, caption = input_ops.parse_sequence_example(
        serialized_sequence_example,
        image_feature=self.config.image_feature_name,
        caption_feature=self.config.caption_feature_name)
    image = self.process_image(encoded_image, thread_id=thread_id)
    images_and_captions.append([image, caption])
```

列表对象 images_and_captions 将作为 input_ops.batch_with_dynamic_pad 内部函数构建队列的输入源,产生训练用 batch 数据包。batch 数据包是 images、input_seqs、target_seqs 和 input_mask 组成的四元组。其中 images 尺寸为[32,299,299,3]表示图像 batch 数据包。input_seqs 表示描述语句的当前描述单词序列。target_seqs 表示描述语句下一个预测描述单词。input_mask 作为掩码用以区分不同长度的描述语句。input_seqs、target_seqs 和 input_mask 的尺寸都为[32,max_{i∈batch}(|S_i|) - 1]。其中, max_{i∈batch}(|S_i|)表示 batch 包中图像描述语句单词数量的最大值。训练的过程就是从输入一次图像特征,再输入一次语句开始标记特征后,依次优化下一个预测单词损失函数值。输入单词和预测单词在描述语句中相差一个位置。

```
# Batch inputs.
queue_capacity = (2 * self.config.num_preprocess_threads *
                  self.config.batch_size)
images, input_seqs, target_seqs, input_mask = (
    input_ops.batch_with_dynamic_pad(images_and_captions,
                                     batch_size=self.config.batch_size,
                                     queue_capacity=queue_capacity))
self.images = images
self.input_seqs = input_seqs
self.target_seqs = target_seqs
self.input_mask = input_mask
```

从如下函数 `batch_with_dynamic_pad` 的关键代码可以看出。若输入的图像摘要列表

对象 `images_and_captions` 中的摘要单词序号值为 $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & - \\ 1 & 2 & 3 & - & - \\ 1 & 2 & - & - & - \end{bmatrix}$ ，则对应的当

前状态单词序号值为 $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & - \\ 1 & 2 & - & - \\ 1 & - & - & - \end{bmatrix}$ ，对应的预测单词序号为 $\begin{bmatrix} 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & - \\ 2 & 3 & - & - \\ 2 & - & - & - \end{bmatrix}$ ，相应

的掩码为 $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$ 。

函数 `batch_with_dynamic_pad` 的关键代码如下所示。关键语句 `tf.train.batch_join` 将在内部开启 `len(enqueue_list)` 个线程，线程 `i` 负责将数据 `enqueue_list[i]` 添加到队列中。这些线程共同维护一个数据队列实现 `batch` 数据包的自动生成。其中，参数 `enqueue_list` 表示为输入的数据源。参数 `batch_size` 表示 `batch` 数据包的大小。参数 `capacity` 表示队列的上限大小。`dynamic_pad` 为真时，要求 `enqueue_list` 中的元素数据维数相同，允许元素的大小尺寸不同。在出队列操作时，尺寸偏小的元素将在右侧被填充。若数据类型为数值则填充 0，若数据类型为字符串则填充空字符串。

```
def batch_with_dynamic_pad(images_and_captions,
                           batch_size,
                           queue_capacity,
                           add_summaries=True):

    enqueue_list = []
    for image, caption in images_and_captions:
        caption_length = tf.shape(caption)[0]
        input_length = tf.expand_dims(tf.sub(caption_length, 1), 0)

        input_seq = tf.slice(caption, [0], input_length)
        target_seq = tf.slice(caption, [1], input_length)
        indicator = tf.ones(input_length, dtype=tf.int32)
        enqueue_list.append([image, input_seq, target_seq, indicator])

    images, input_seqs, target_seqs, mask = tf.train.batch_join(
        enqueue_list,
        batch_size=batch_size,
        capacity=queue_capacity,
        dynamic_pad=True,
```

```

        name="batch_and_pad")
    return images, input_seqs, target_seqs, mask

```

图像向量化

经过前期的数据操作后，图像数据被转换为尺寸为[32,299,299,3]的张量 batch 数据包。而实际上，TensorFlow 内部已经实现了多种经典网络的完整实现，其中包括了 inception_v3 网络。以下代码概括了 inception_v3 网络的操作定义。inception_v3_base 函数返回值 net 表示网络的最顶端输出层结果，而 end_points 则包括了整个网络的相关信息。

```

.....
from tensorflow.contrib.slim.python.slim.nets.inception_v3 import
inception_v3_base
.....
net, end_points = inception_v3_base(images, scope=scope)
.....

```

在 TensorFlow 源码 tensorflow\contrib\slim\python\slim\nets\inception_v3.py 文件内部可以看到 inception_v3_base 函数的内部实现。从内部实现中可以看出 inception_v3 网络主要也是由 tensorflow.contrib.layers 模块构建的。以下代码片段显示了使用 tensorflow.contrib.layers 模块构建深度神经网络。函数 variable_scope.variable_scope 定义变量命名前缀以便复用训练好的模型参数，并验证张量列表[inputs]中的元素与当前操作定义来自于同一个网络。函数 arg_scope 重定义了参数列表[layers.conv2d, layers_lib.max_pool2d, layers_lib.avg_pool2d] 指定函数的默认参数。类函数 layers.conv2d 定义了卷积操作，并返回操作结果张量。关于 layers 模块中实现的其他类型网络层操作定义可查看源码文件 tensorflow\contrib\layers\python\layers\layers.py。

```

from tensorflow.contrib import layers
.....
with variable_scope.variable_scope(scope, 'InceptionV3',
[inputs]):
    with arg_scope(
        [layers.conv2d, layers_lib.max_pool2d, layers_lib.avg_
pool2d],
        stride=1,
        padding='VALID'):
        # 299 x 299 x 3
        end_point = 'Conv2d_1a_3x3'

```



```

        net = layers.conv2d(inputs, depth(32), [3, 3], stride=2,
scope=end_point)
        end_points[end_point] = net
        if end_point == final_endpoint:
            return net, end_points
        # 149 x 149 x 32
        end_point = 'Conv2d_2a_3x3'
        net = layers.conv2d(net, depth(32), [3, 3], scope=end_point)
        end_points[end_point] = net
        if end_point == final_endpoint:
            return net, end_points
        # 147 x 147 x 32 return net, end_points
.....

```

定义了 inception_v3 网络操作后还需要实现对训练模型的加载。模型加载代码如下所示，函数 `tf.get_collection` 定义操作返回指定命名的变量列表。`tf.train.Saver` 根据变量列表构建存储对象。最后，`saver.restore` 函数根据网络会话对象 `sess` 和模型文件存储路径定义 inception_v3 网络的训练参数还原操作。最后，定义的模型初始化函数 `self.init_fn` 将作为参数输入到网络的训练函数中。

```

        self.inception_variables = tf.get_collection(tf.GraphKeys.GLOBAL_
VARIABLES, scope="InceptionV3")
.....
        saver = tf.train.Saver(self.inception_variables)
.....
        def restore_fn(sess):
            saver.restore(sess, self.config.inception_checkpoint_file)
        self.init_fn = restore_fn

```

inception_v3 网络是 NIC 算法图像向量化的主要组成部分。输入尺寸为 [32,299,299,3] 的 batch 数据训练包，经过 inception_v3 网络后将得到尺寸为 [32,8,8,2048] 大小的 batch 数据包特征描述矩阵。即 32 张尺寸为 [299,299,3] 大小的图像被分别抽象化为 32 个 [8,8,2048] 大小的矩阵描述。为了强化特征的平移不变性，在 inception_v3 网络基础上还有一个尺寸为 [8,8] 的均值池化层，强化特征的平移不变性。最后，在网络添加一个尺寸为 [2048,512] 的全连接层，将图像特征转化为 512 维的向量化的表示。

单词向量化

实际上单词向量化的过程非常简单。这一过程只是使用 `tf.get_variable` 函数获取一个尺寸为 `[self.config.vocab_size, self.config.embedding_size]` 的可训练的张量变量，用以表示整个单词集合的向量化表示。然后 `tf.nn.embedding_lookup` 函数定义操作返回当前描述单词序号对应的向量子集。

```
with tf.variable_scope("seq_embedding"), tf.device("/cpu:0"):
    embedding_map = tf.get_variable(
        name="map",
        shape=[self.config.vocab_size,
self.config.embedding_size],
        initializer=self.initializer)
    seq_embeddings = tf.nn.embedding_lookup(embedding_map,
self.input_seqs)
```

```
self.seq_embeddings = seq_embeddings
```

从 `inference_wrapper_base.py` 文件可知，在预测阶段，NIC 算法调用了 `tf.train.latest_checkpoint` 函数返回最后保存的模型文件绝对路径，并通过 `saver.restore` 函数在当前会话中加载变量字典。因此，在训练阶段 `tf.get_variable` 函数将创建 `embedding_map` 张量变量。而在预测阶段，该函数将使用训练好的模型参数。

```
def _create_restore_fn(self, checkpoint_path, saver):
    if tf.gfile.IsDirectory(checkpoint_path):
        checkpoint_path = tf.train.latest_checkpoint(checkpoint_path)
    if not checkpoint_path:
        raise ValueError("No checkpoint file found in: %s" % checkpoint_
path)
```

```
def _restore_fn(sess):
    tf.logging.info("Loading model from checkpoint: %s",
checkpoint_path)
    saver.restore(sess, checkpoint_path)
    tf.logging.info("Successfully loaded checkpoint: %s",
os.path.basename(checkpoint_path))
    return _restore_fn
```

`saver` 为模型的保存读取提供了十分简便的函数接口。训练时结合 `tf.get_variable` 函数，更可使用 `saver` 实现二次开机后的持续训练。需要注意的是，`tf.train.latest_`

checkpoint 函数将会读取训练模型文件夹的模型文件存放路径。而该文件的路径是按照绝对路径保存的。因此若更改了保存模型的存放位置后还需要修改 checkpoint 文件中的路径信息才能正常使用 tf.train.latest_checkpoint 函数。

定义模型结构操作

NIC 算法的模型结构在训练时需要同时根据图像特征和单词向量定义损失函数并计算预测误差,而在预测时则只需要根据图像内容由训练模型推算出最贴切的描述语句。因此,网络的结构定义在训练时和预测时是不同的。在程序的内部实现中可以使用条件判断语句定义不同的网络结构。

在 show_and_tell_model.py 文件中的 build_model 函数可以看到 LSTM 网络结构的实现程序。首先定义 tf.nn.rnn_cell.BasicLSTMCell 类的对象 lstm_cell。BasicLSTMCell 表示 TensorFlow 中的基础 LSTM 网络。lstm_cell 被定义为拥有 self.config.num_lstm_units 个特征长度的网络节点,以元组形式返回预测结果和状态值。仅在训练时使用 tf.nn.rnn_cell.DropoutWrapper 类对 lstm_cell 网络重构一个带有 Dropout 功能的节点。

```
lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(
    num_units=self.config.num_lstm_units, state_is_tuple=True)
if self.mode == "train":
    lstm_cell = tf.nn.rnn_cell.DropoutWrapper(
        lstm_cell,
        input_keep_prob=self.config.lstm_dropout_keep_prob,
        output_keep_prob=self.config.lstm_dropout_keep_prob)
```

随后定义 LSTM 网络的变量作用域“lstm”,并令其内部各网络参数按照类函数 self.initializer 实现的均匀分布方式初始生成随机参数。在预测时,程序将根据变量作用域名称自动加载训练好的模型参数。因网络是通过 batch 数据包方式训练的,所以还需要使用 lstm_cell.zero_state 函数创建输入图像特征时对应的零状态元组信息 zero_state,作为初始时的 LSTM 状态值。在训练时,默认 batch 包中含有 32 个样本,而在预测时 batch 包中的样本数量为 1。接下来的函数 lstm_scope.reuse_variables() 确保 lstm_cell 中的网络状态参数将被循环更新使用。

```
with tf.variable_scope("lstm", initializer=self.initializer)
as lstm_scope:
```

```

# Feed the image embeddings to set the initial LSTM state.
zero_state = lstm_cell.zero_state(
    batch_size=self.image_embeddings.get_shape()[0],
    dtype=tf.float32)
_, initial_state = lstm_cell(self.image_embeddings, zero_state)

# Allow the LSTM variables to be reused.
lstm_scope.reuse_variables()

```

在预测时定义一个张量连接操作命令，名为“initial_state”，用以在每张图像的预测起始阶段通过该操作创建并返回一个初始状态张量。state_feed 表示含有上一刻的预测结果和状态信息的占位符。state_tuple 是占位符 state_feed 的元组表示形式。lstm_cell 网络训练时要求输入特征为 2 维，第一维表示样本，第二维表示特征。不同于训练时 self.seq_embeddings 拥有描述语句单词的向量化结果，预测时 self.seq_embeddings 仅对单张图像的描述单词序列进行预测。此时，使用 tf.squeeze 函数对数据维数进行压缩确保输入有效数据。最后，定义名称为“state”的操作将状态元组信息转换为张量信息。产生的 lstm_outputs 张量表示当前预测单词的特征信息，默认时尺寸最大为[3,512]。因为默认时，将按照贪心算法，仅保留单词置信度最大的 3 个预测单词序列。

```

if self.mode == "inference":
    # In inference mode, use concatenated states for convenient
    # feeding and fetching.
    tf.concat(1, initial_state, name="initial_state")

    # Placeholder for feeding a batch of concatenated states.
    state_feed = tf.placeholder(dtype=tf.float32,
                                shape=[None,
                                        sum(lstm_cell.state_size)],
                                name="state_feed")
    state_tuple = tf.split(1, 2, state_feed)

    # Run a single LSTM step.
    lstm_outputs, state_tuple = lstm_cell(
        inputs=tf.squeeze(self.seq_embeddings,
                           squeeze_dims=[1]),
        state=state_tuple)

    # Concatenate the resulting state.
    tf.concat(1, state_tuple, name="state")

```

在训练时，基于定义的单个 LSTM 网络节点，根据图像特征在零状态下产生的初始状态，依次向网络中输入单词向量化序列，以产生序列化的预测结果。其中张量 `sequence_length` 表示 batch 训练包对应样本描述语句的单词数量。函数 `tf.nn.dynamic_rnn` 专门针对序列化的循环网络提供 batch 包式的预测结果。`lstm_outputs` 表示 batch 单词序列的预测特征信息，尺寸为 $[32, N - 1, 512]$ 。其中， N 表示 batch 包样本中描述语句的最大单词数量。因为需要使用 1 个偏移位置预测下一个单词，所以序列的样本标签数量为 $N - 1$ 。

```

else:
    # Run the batch of sequence embeddings through the LSTM.
    sequence_length = tf.reduce_sum(self.input_mask, 1)
    lstm_outputs, _ = tf.nn.dynamic_rnn(cell=lstm_cell,
                                         inputs=self.seq_embeddings,
                                         sequence_length=sequence_length,
                                         initial_state=initial_state,
                                         dtype=tf.float32,
                                         scope=lstm_scope)

```

与基于语句计算均值的损失函数设计相比，基于单词预测均值的损失函数设计方式拥有更好的性能。这是因为在预测时仅有图像输入数据，单词输入信息是由算法自身产生并再次输入到算法中的。而训练时，单词信息的输入直接来自于真值描述语句。基于单个单词的预测均值损失函数能够更好地表征训练和预测过程。而基于语句的损失函数将导致训练过程与预测过程存在一定的偏离差异。

无论是训练时或预测时，上述 `lstm_outputs` 预测单词的特征还需要连接一个全连接层，才能实现对单词特征的分类预测。为了将模型优化问题转换为上下文相关的序列特征单词预测问题。在进入全连接层之前，还需要对特征张量进行 2 维转换确保有效输入。`tf.reshape` 语句将 `lstm_outputs` 的维度从 3 维转换为 2 维，即原先是以 [图像样本序号，描述语句单词序号，特征序号] 尺寸转换为 [单词序号，特征序号] 的形式。在全连接层输入的 batch 单词样本数量是输入 `tf.nn.dynamic_rnn` 函数时 batch 样本描述语句单词的数量总和。全连接层返回的 logits 分类置信度值，在预测阶段将被用于选择分类结果，而在训练阶段将被用于计算损失函数。

```

# Stack batches vertically.

```

```
lstm_outputs = tf.reshape(lstm_outputs, [-1, lstm_cell.output_size])
```

```
with tf.variable_scope("logits") as logits_scope:
    logits = tf.contrib.layers.fully_connected(
        inputs=lstm_outputs,
        num_outputs=self.config.vocab_size,
        activation_fn=None,
        weights_initializer=self.initializer,
        scope=logits_scope)
```

在预测时，定义操作“softmax”便于获取单词的预测分类结果。

```
if self.mode == "inference":
    tf.nn.softmax(logits, name="softmax")
```

而在训练时，将根据 batch 数据的所有单词序列预测结果计算损失函数值。因为不同图像描述语句长度不同，在具体的计算过程中使用了 self.input_mask 作为有效单词的掩码矩阵 weights。函数 tf.contrib.losses.add_loss 将根据 batch 损失函数估计全局损失函数 total_loss。从图的定义可以看出 total_loss 中仅含有 1 个 batch_loss 节点。当模型中定义了多个损失函数节点时使用该函数可以返回一个总的损失函数值。训练时加入 tf.scalar_summary 等统计函数将有利于 TensorBoard 的可视化分析，便于验证模型的有效性。

```
else:
    targets = tf.reshape(self.target_seqs, [-1])
    weights = tf.to_float(tf.reshape(self.input_mask, [-1]))

    # Compute losses.
    losses = tf.nn.sparse_softmax_cross_entropy_with_logits(logits,
targets)
    batch_loss = tf.div(tf.reduce_sum(tf.mul(losses, weights)),
                        tf.reduce_sum(weights),
                        name="batch_loss")
    tf.contrib.losses.add_loss(batch_loss)
    total_loss = tf.contrib.losses.get_total_loss()

    # Add summaries.
    tf.scalar_summary("batch_loss", batch_loss)
    tf.scalar_summary("total_loss", total_loss)
    for var in tf.trainable_variables():
```

```
tf.histogram_summary(var.op.name, var)

self.total_loss= total_loss

.....
```

神经网络的运行

上述代码实现了预测网络和训练网络的操作定义，但并没有基于 TensorFlow 会话的 Session 运行语句。训练网络的运行语句相对简单，只需要根据损失函数的返回值定义训练操作，再运行训练操作即可。从 train.py 文件可知关键代码如下，函数 tf.contrib.layers.optimize_loss 输入损失函数值 model.total_loss、全局学习步长 model.global_step、初始学习率 learning_rate、优化函数类型、梯度裁剪越界操作和学习率更新函数，返回损失函数优化操作。其中，学习率更新函数 learning_rate_decay_fn 在内部调用了 TensorFlow 函数 tf.train.exponential_decay，对应的是一个指数衰减学习率更新操作。该函数的输入参数由 model.global_step 和 learning_rate 共同组成。最后，启用 tf.contrib.slim.learning.train 函数开始训练。该函数必须指定训练优化操作、模型和日志文件存放路径。在训练耗时较长的大数据时，自定义 tf.train.Saver 限制训练过程中保存模型的最大数量是十分必要的。

```
# Set up the training ops.
train_op = tf.contrib.layers.optimize_loss(
    loss=model.total_loss,
    global_step=model.global_step,
    learning_rate=learning_rate,
    optimizer=training_config.optimizer,
    clip_gradients=training_config.clip_gradients,
    learning_rate_decay_fn=learning_rate_decay_fn)

# Set up the Saver for saving and restoring model checkpoints.
saver = tf.train.Saver(max_to_keep=training_config.max_
checkpoints_to_keep)

# Run training.
tf.contrib.slim.learning.train(
    train_op,
    train_dir,
    log_every_n_steps=FLAGS.log_every_n_steps,
    graph=g,
    global_step=model.global_step,
```

```

number_of_steps=FLAGS.number_of_steps,
init_fn=model.init_fn,
saver=saver)

```

从 `run_inference.py` 文件可知,在对单张图像内容的摘要进行预测时主要是通过调用 `inference_utils\caption_generator.py` 文件的类函数 `CaptionGenerator.beam_search` 实现的。

`beam_search` 函数首先使用函数 `self.model.feed_image` 在内部运行会话操作“`initial_state`”获取图像数据在 LSTM 网络中的零输入初始状态特征。返回列表对象 `initial_state`,因只输入一个图像,所以 `initial_state` 只有一个列表元素。初始化摘要类型 `Caption` 对象 `initial_beam`,对应语句 `sentence` 含有一个序号为 1 的(`self.vocab.start_id`)表示语句开始特殊标记的单词元素。初始状态张量为 `initial_state[0]`。该语句的平均预测概率 `logprob` 和得分 `score` 均为 0。

```

def beam_search(self, sess, encoded_image):
    initial_state = self.model.feed_image(sess, encoded_image)

    initial_beam = Caption(
        sentence=[self.vocab.start_id],
        state=initial_state[0],
        logprob=0.0,
        score=0.0,
        metadata=[""])

```

定义类型为 `TopN` 的对象 `partial_captions`,用于保存当前预测概率最大的 `self.beam_size` 种可能的预测单词序列。初始预测结果 `initial_beam` 被加入到 `partial_captions` 中。类型为 `TopN` 的对象 `complete_captions` 负责保存最终预测概率最大的 `self.beam_size` 种摘要预测语句。类型为 `TopN` 的对象在进行 `push` 操作时,若当前队列元素数量小于 `self.beam_size` 则直接入队列,否则将会根据元素的重载函数 `__cmp__` 判断,确保入队列元素置信度大于队列元素,并进行移除队列操作,以确保队列长度始终小于 `self.beam_size`。

```

partial_captions = TopN(self.beam_size)
partial_captions.push(initial_beam)
complete_captions = TopN(self.beam_size)

```

根据配置参数限制描述语句的最大长度进行描述语句的单词预测。函数

`partial_captions.extract` 返回类型全部为 `Caption` 的列表对象。使用函数 `partial_captions.reset` 清空 `partial_captions` 对象,以便保存下一刻预测概率最大的前 `self.beam_size` 种可能的预测单词序列。`self.model.inference_step` 函数内部将根据当前预测语句的最后一个单词,确定预测模型单词向量输入占位符"input_feed"操作。类似的,基于当前语句状态信息,确定预测模型状态输入占位符"initial_state"操作。`self.model.inference_step` 函数内部将运行网络定义操作"softmax:0"和"lstm/state:0"返回输入单词的预测结果 `softmax` 和状态信息 `new_states`。

```
# Run beam search.
for _ in range(self.max_caption_length - 1):
    partial_captions_list = partial_captions.extract()
    partial_captions.reset()
    input_feed = np.array([c.sentence[-1] for c in partial_
captions_list])
    state_feed = np.array([c.state for c in partial_captions_
list])

    softmax, new_states, _ = self.model.inference_step(sess,
                                                    input_feed,
                                                    state_feed)
```

依次遍历 `partial_captions_list` 中的所有预测单词序列结果。初始时 `partial_captions_list` 只有 1 个含有 1 个单词的摘要语句。往后的循环中最多只有 `self.beam_size` 个语句。对于每一个已产生的单词序列,考虑其置信度最大的前 `self.beam_size` 个单词预测结果保存至 `words_and_probs` 中。实际上,为了搜索置信度最大的前 `self.beam_size` 个摘要语句,在单词预测时对每一个语句考虑前 `self.beam_size` 个单词预测结果来近似达到最优目标。

```
for i, partial_caption in enumerate(partial_captions_list):
    word_probabilities = softmax[i]
    state = new_states[i]
    # For this partial caption, get the beam_size most probable
next words.
    words_and_probs = list(enumerate(word_probabilities))
    words_and_probs.sort(key=lambda x: -x[1])
    words_and_probs = words_and_probs[0:self.beam_size]
```

若预测置信度大于阈值 $1e-12$,将预测单词加入到当前单词预测序列 `sentence` 中。

若当前预测单词 w 为结束符单词, 则尝试构建 Caption 并将其加入到 `complete_captions` 中。 `complete_captions.push` 函数将根据元素的置信度决定是否真正加入。若 w 不为结束符单词, 将合并后的语句和状态信息再次添加到 `partial_captions` 对象中, 以进行下一次预测。若 `partial_captions` 对象为空, 则说明单词序列已经全部以结束符单词预测完毕, 提前退出搜索。在计算单词置信度时, 若变量 `self.length_normalization_factor` 大于 0 则置信度为预测单词的平均置信度, 否则表示当前预测单词的置信度。

```
# Each next word gives a new partial caption.
for w, p in words_and_probs:
    if p < 1e-12:
        continue # Avoid log(0).
    sentence = partial_caption.sentence + [w]
    logprob = partial_caption.logprob + math.log(p)
    score = logprob

    if w == self.vocab.end_id:
        if self.length_normalization_factor > 0:
            score /= len(sentence)**self.length_normalization_factor
        beam = Caption(sentence, state, logprob, score,
metadata_list)
        complete_captions.push(beam)
    else:
        beam = Caption(sentence, state, logprob, score,
metadata_list)
        partial_captions.push(beam)
    if partial_captions.size() == 0:
        # We have run out of partial candidates; happens when beam_size
= 1.
        Break
```

若单词序列预测过程中没有预测结束符单词, `complete_captions` 元素为 0, 返回 `partial_captions` 临时预测单词序列。否则, 对 `complete_captions` 对象数据置信度进行降排序返回。

```
if not complete_captions.size():
    complete_captions = partial_captions

return complete_captions.extract(sort=True)
```

最后, 可从 `run_inference.py` 文件了解到, 通过以下代码将单词序号转换为语句

进行输出。

```
for i, caption in enumerate(captions):
    # Ignore begin and end words.
    sentence = [vocab.id_to_word(w) for w in
caption.sentence[1:-1]]
    sentence = " ".join(sentence)
    print(" %d) %s (p=%f)" % (i, sentence,
math.exp(caption.logprob)))
```

6.2.4 NIC 算法的实验数据与结论

语句相似性评价方法

目前图像摘要问题并无客观严谨的评价标准,即使是人工判断亦存在差异。该问题尚无统一评价标准。在机器翻译领域中,为了有效利用人工标记语句评估算法性能,人们提出了很多参考评价方法。相关评价方法如表 6-7 所示:

表 6-7 图像摘要算法评价方法

名称	概要
BLUE	通过位置无关的词组比较统计,返回字符串语句之间的相似度得分。所有参考语句产生 1 个相似得分。 KishorePapineni 等于 2002 年在论文 <i>BLEU: a Method for Automatic Evaluation of Machine-Translation</i> 中提出
ROUGE	使用位置相关的子字符串比较统计方法计算字符串语句之间的相似度得分。每个参考语句产生 1 个得分。 C.-Y. Lin 等于 2004 年在论文 <i>Rouge: A package for automatic evaluation of summaries</i> 中提出
METEOR	以单词为基本单元进行 1 对 1 的比较统计。根据单词配对的位置关系,从多个参考语句中选择位置上错位次序最小的参考语句计算两语句的相似度得分。每个参考语句产生 1 个得分。 S. Banerjee 等于 2005 年在论文 <i>Meteor: An automatic metric for machine translation evaluation with improved correlation with human judgments</i> 中提出
CIDEr	对参考描述语句进行连续单词短语的词频统计。基于此,对语句使用向量化表示。最后使用向量内积相关公式计算语句的相似度。 R. Vedantam 等于 2015 年在论文 <i>CIDEr: Consensus-Based image description evaluation</i> 中提出

BLUE 评价方法首先统计语句中长度为 n 的词组 n_gram 的出现次数 $count(n_gram)$ 。令多个人工标注的参考语句构成集合 R , 待评价语句为 c 。对于 c 中

若当前预测单词 w 为结束符单词, 则尝试构建 Caption 并将其加入到 `complete_captions` 中。 `complete_captions.push` 函数将根据元素的置信度决定是否真正加入。若 w 不为结束符单词, 将合并后的语句和状态信息再次添加到 `partial_captions` 对象中, 以进行下一次预测。若 `partial_captions` 对象为空, 则说明单词序列已经全部以结束符单词预测完毕, 提前退出搜索。在计算单词置信度时, 若变量 `self.length_normalization_factor` 大于 0 则置信度为预测单词的平均置信度, 否则表示当前预测单词的置信度。

```
# Each next word gives a new partial caption.
for w, p in words_and_probs:
    if p < 1e-12:
        continue # Avoid log(0).
    sentence = partial_caption.sentence + [w]
    logprob = partial_caption.logprob + math.log(p)
    score = logprob

    if w == self.vocab.end_id:
        if self.length_normalization_factor > 0:
            score /= len(sentence)**self.length_normalization_factor
        beam = Caption(sentence, state, logprob, score,
            metadata_list)
        complete_captions.push(beam)
    else:
        beam = Caption(sentence, state, logprob, score,
            metadata_list)
        partial_captions.push(beam)
    if partial_captions.size() == 0:
        # We have run out of partial candidates; happens when beam_size
        = 1.
        Break
```

若单词序列预测过程中没有预测结束符单词, `complete_captions` 元素为 0, 返回 `partial_captions` 临时预测单词序列。否则, 对 `complete_captions` 对象数据置信度进行降排序返回。

```
if not complete_captions.size():
    complete_captions = partial_captions

return complete_captions.extract(sort=True)
```

最后, 可从 `run_inference.py` 文件了解到, 通过以下代码将单词序号转换为语句

进行输出。

```
for i, caption in enumerate(captions):
    # Ignore begin and end words.
    sentence = [vocab.id_to_word(w) for w in
caption.sentence[1:-1]]
    sentence = " ".join(sentence)
    print("%d) %s (p=%f)" % (i, sentence,
math.exp(caption.logprob)))
```

6.2.4 NIC 算法的实验数据与结论

语句相似性评价方法

目前图像摘要问题并无客观严谨的评价标准,即使是人工判断亦存在差异。该问题尚无统一评价标准。在机器翻译领域中,为了有效利用人工标记语句评估算法性能,人们提出了很多参考评价方法。相关评价方法如表 6-7 所示:

表 6-7 图像摘要算法评价方法

名称	概要
BLUE	通过位置无关的词组比较统计,返回字符串语句之间的相似度得分。所有参考语句产生 1 个相似得分。 KishorePapineni 等于 2002 年在论文 <i>BLEU: a Method for Automatic Evaluation of Machine-Translation</i> 中提出
ROUGE	使用位置相关的子字符串比较统计方法计算字符串语句之间的相似度得分。每个参考语句产生 1 个得分。 C.-Y.Lin 等于 2004 年在论文 <i>Rouge: A package for automatic evaluation of summaries</i> 中提出
METEOR	以单词为基本单元进行 1 对 1 的比较统计。根据单词配对的位置关系,从多个参考语句中选择位置上错位次序最小的参考语句计算两语句的相似度得分。每个参考语句产生 1 个得分。 S.Banerjee 等于 2005 年在论文 <i>Meteor: An automatic metric for machine evaluation with improved correlation with human judgments</i> 中提出
CIDEr	对参考描述语句进行连续单词短语的词频统计。基于此,对语句使用向量化表示。最后使用向量内积相关公式计算语句的相似度。 R.Vedantam 等于 2015 年在论文 <i>CIDEr: Consensus-Based image description evaluation</i> 中提出

BLUE 评价方法首先统计语句中长度为 n 的词组 n_gram 的出现次数 $count(n_gram)$ 。令多个人工标注的参考语句构成集合 R , 待评价语句为 c 。对于 c 中

若当前预测单词 w 为结束符单词, 则尝试构建 `Caption` 并将其加入到 `complete_captions` 中。`complete_captions.push` 函数将根据元素的置信度决定是否真正加入。若 w 不为结束符单词, 将合并后的语句和状态信息再次添加到 `partial_captions` 对象中, 以进行下一次预测。若 `partial_captions` 对象为空, 则说明单词序列已经全部以结束符单词预测完毕, 提前退出搜索。在计算单词置信度时, 若变量 `self.length_normalization_factor` 大于 0 则置信度为预测单词的平均置信度, 否则表示当前预测单词的置信度。

```
# Each next word gives a new partial caption.
for w, p in words_and_probs:
    if p < 1e-12:
        continue # Avoid log(0).
    sentence = partial_caption.sentence + [w]
    logprob = partial_caption.logprob + math.log(p)
    score = logprob

    if w == self.vocab.end_id:
        if self.length_normalization_factor > 0:
            score /= len(sentence)**self.length_normalization_factor
        beam = Caption(sentence, state, logprob, score,
            metadata_list)
        complete_captions.push(beam)
    else:
        beam = Caption(sentence, state, logprob, score,
            metadata_list)
        partial_captions.push(beam)
    if partial_captions.size() == 0:
        # We have run out of partial candidates; happens when beam_size
        = 1.
        Break
```

若单词序列预测过程中没有预测结束符单词, `complete_captions` 元素为 0, 返回 `partial_captions` 临时预测单词序列。否则, 对 `complete_captions` 对象数据置信度进行降排序返回。

```
if not complete_captions.size():
    complete_captions = partial_captions

return complete_captions.extract(sort=True)
```

最后, 可从 `run_inference.py` 文件了解到, 通过以下代码将单词序号转换为语句

进行输出。

```
for i, caption in enumerate(captions):
    # Ignore begin and end words.
    sentence = [vocab.id_to_word(w) for w in
caption.sentence[1:-1]]
    sentence = " ".join(sentence)
    print("%d) %s (p=%f)" % (i, sentence,
math.exp(caption.logprob)))
```

6.2.4 NIC 算法的实验数据与结论

语句相似性评价方法

目前图像摘要问题并无客观严谨的评价标准,即使是人工判断亦存在差异。该问题尚无统一评价标准。在机器翻译领域中,为了有效利用人工标记语句评估算法性能,人们提出了很多参考评价方法。相关评价方法如表 6-7 所示:

表 6-7 图像摘要算法评价方法

名称	概要
BLUE	通过位置无关的词组比较统计,返回字符串语句之间的相似度得分。所有参考语句产生 1 个相似得分。 KishorePapineni 等于 2002 年在论文 <i>BLEU:aMethodforAutomaticEvaluationofMachine-Translation</i> 中提出
ROUGE	使用位置相关的子字符串比较统计方法计算字符串语句之间的相似度得分。每个参考语句产生 1 个得分。 C.-Y.Lin 等于 2004 年在论文 <i>Rouge:Apakageforautomaticevaluationofsum-maries</i> 中提出
METEOR	以单词为基本单元进行 1 对 1 的比较统计。根据单词配对的位置关系,从多个参考语句中选择位置上错位次序最小的参考语句计算两语句的相似度得分。每个参考语句产生 1 个得分。 S.Banerjee 等于 2005 年在论文 <i>Meteor:Anautomaticmetricformtevaluationwithimproved-correlationwithhumanjudgments</i> 中提出
CIDEr	对参考描述语句进行连续单词短语的词频统计。基于此,对语句使用向量化表示。最后使用向量内积相关公式计算语句的相似度。 R.Vedantam 等于 2015 年在论文 <i>CIDEr:Consensus-Basedimagedescriptionevaluation</i> 中提出

BLUE 评价方法首先统计语句中长度为 n 的词组 n_gram 的出现次数 $count(n_gram)$ 。令多个人工标注的参考语句构成集合 R , 待评价语句为 c 。对于 c 中

任意一个长度为 n 的词组 n_gram , 定义该词组的匹配次数为 $count_{clip}(n_gram)$ 。则语句 c 的所有词组 n_gram 的匹配相似度为 p_n 。计算公式如下所示:

$$count_{clip}(n_gram) = \min(\max_{r \in R}(count_r(n_gram)), count_c(n_gram))$$

$$p_n = \frac{\sum_{n_gram \in c} count_{clip}(n_gram)}{\sum_{n_gram \in c} count_c(n_gram)}$$

例如, 待评价语句 “the the the the the the the” 和两个参考语句 “The cat is on the mat” “There is a cat on the mat”。当 $n=1$ 时, 仅有单词 “the” 存在匹配, 且在第一个参考语句中匹配次数最大为 2, 第二个语句的匹配次数为 1。此时, 选择匹配最大匹配值参与计算, $p_1 = \frac{2}{7}$ 。当 $n=2$ 时, 待评价语句中的 “the the” 出现的次数为 6, 而参考语句中最大出现次数为 0。此时 $p_2 = \frac{0}{6}$ 。显然, 当 n 取值较小时匹配次数肯定大于 n 取值较大的匹配次数。 n 越大能够度量的语义确定性越大, 但度量能力越弱。另外, 仅使用 p_n 统计匹配次数, 无法度量待评价语句的冗余信息。考虑另外一个例子, 两个待评价机器翻译语句 “I always invariably perpetually do.” 和 “I always do.”。与之对应有三个参考语句为 “I always do.”、“I invariably do.”、“I perpetually do.”。两个待评价语句 p_1 值相同, 且第一个待评价语句中含有更多的参考语句词汇。但实际上, 第二个待评价语句更加言简意赅, 第一个待评价语句的单词分散在多个参考语句中。对此, BLUE 方法还提出使用基于语句长度的惩罚因素 BP。计算公式如下, 其中 c 表示待评价语句长度, r 表示参考语句中语句长度与之最相似的语句长度。 N 表示度量词组的最大长度, 建议值为 4。

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

$$BLUE = BP \cdot \exp\left(\frac{1}{N} \sum_{n=1}^N \log p_n\right)$$

BLUE 方法计算简单, 在机器翻译领域使用较为广泛。但该方法使用的是与位置无关的单词匹配统计, 难以度量与词序相关的语义信息。例如, 令两个待评价语句为 “police kill the gunman” 和 “the gunman kill police”。若对应的参考语句为 “police killed

the gunman”。两待评价语句含义完全不同，但它们的 BLUE 值相同。对此，ROUGE 方法和 METEOR 方法分别对其进行了改进。

ROUGE 方法定义最长共序子序列操作来度量字符串语句的匹配程度。令序列 $Y = [y_1, y_2, \dots, y_n]$ 与序列 $X = [x_1, x_2, \dots, x_m]$ 的共序子序列为 $Z = [z_1, z_2, \dots, z_k]$ 。若 Z 不为空，则一定存在一个升序的整数序列 $[i_1, i_2, \dots, i_k]$ ，使得 $y_{o+l} = z_l = x_{i_l}$ 。其中 $o \in [0, n-k]$ 、 $l \in [1, k]$ 、 $i_l \in [1, m]$ 。当 k 取最大值时，为最大共序子字符串。定义最大共序子序列串长度函数为 LCS。对于长度为 m 的待评价语句 c 和长度为 n 的参考语句 r ，可按照如下公式计算语句的匹配精度 P 和覆盖率 R ，从而进一步得出 ROUGE 的评价值。其中， β 表示权重参数，建议值为 1。对于之前“police killed the gunman”例子，两个待评价语句的 $LCS(X, Y)$ 分别为 0.75 和 0.5，ROUGE 方法的评价值分别为 0.75 和 0.5。因此，第一个待评价语句更高。ROUGE 方法的计算公式如下。

$$R = \frac{LCS(X, Y)}{m}$$

$$P = \frac{LCS(X, Y)}{n}$$

$$ROUGE = \frac{(1 + \beta^2)RP}{R + \beta^2 P}$$

METEOR 方法提出基于单词的 1 对 1 配对比较方法。输入两个字符串语句，METEOR 方法进行 3 次单词的 1 对 1 配对。第一次配对时，仅配对拼写完全相同的单词。第二次配对允许配对拼写不同但词根相同的单词，例如“kill”和“killed”等。第三次配对时允许配对同义词。当配对过程中存在多种可能性时，最后使用动态规划算法选择配对位置关系“交叉”次数最少的配对情况。例如在第一次配对时待评价语句出现了 x 次单词“apple”，而参考语句出现了 y 次单词“apple”。则总共可能存在 $C_{\max(x, y)}^{\min(x, y)}$ 种匹配可能。对于语句 c 和 r 中两对可能的匹配单词 (c_i, r_j) 和 (c_k, r_l) ，若这些单词在位置上满足公式 $(\text{pos}(c_i) - \text{pos}(c_k)) \times (\text{pos}(r_l) - \text{pos}(r_j)) < 0$ 则表示它们的配对存在位置上的“交叉”。完成一次配对选择后，进入下一次配对时将不再使用之前已经配对过的单词。根据单词的匹配结果得出一对语句的相似度 F_{mean} 。计算公式如下。其中， R 、 P 的计算与 ROUGE 方法类似。只是这里使用的单词匹配值。

$$Fmean = \frac{10RP}{R + 9P}$$

此外,还要根据语句单词的配对情况对语句进行分块。分析两语句的语序相关性。配对单词在源语句和配对语句中的位置相邻且相对位置保持不变的集合组成一个分块 chunks。从图 6-10 可以看出对于上一个例子的分块结果。图 6-10 左图被的分块 chunks 数量为 1, 右图的分块数量 chunks 为 3。

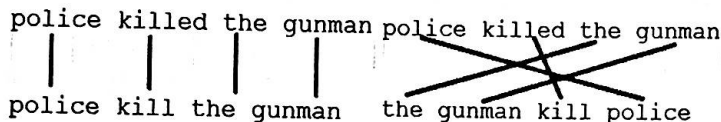


图 6-10 METEOR 方法单词配对分块示意图

METEOR 方法的度量值由语句连贯性惩罚值 Penalty 和单词匹配度量值 Fmean 共同决定。chunks_num 表示匹配后的分块数量。match_num 表示单词匹配数量。对于之前“police killed the gunman”的例子,在 METEOR 方法中 Fmean 都为 1。但 Penalty 值分别为 0.9921875 和 0.7890265。因此, METEOR 方法的评价值分别为 0.9921875 和 0.7890265。特别的,当单词匹配次数为 0 时,令 Penalty = 0.5, METEOR = 0。

$$Penalty = 0.5 \times \left(\frac{chunks_num}{match_num} \right)^3$$

$$METEOR = Fmean \times (1 - Penalty)$$

与之前介绍的三种机器翻译评价方法相比较, CIDEr 方法是专门针对图像摘要问题提出的评价方法。它的特点是统计所有图像摘要样本的词频信息,并基于此构建语句的特征向量。通过度量特征向量来表示两个图像摘要语句的相似程度 $CIDEr(c_i, S_i)$ 。相关计算公式如下。其中, c_i 表示待评价语句, S_i 表示对应的参考语句集合。 S_i 可能由 N 条语句 s_{ij} 构成。在 MSCOCO 数据集中一张图像对应 5 条参考描述语句。 $g^n(c_i)$ 和 $g^n(s_{ij})$ 分别表示待评价语句和参考语句的特征向量。它的每一个分量由长度为 n 的对应词组 k 由公式 $g_k(c_i)$ 和 $g_k(s_{ij})$ 计算得出。 $h_k(s_{ij})$ 表示词组 k 在语句 s_{ij} 中的计数统计。 Ω 表示长度为 n 的词组词典集合。 $\sum_{w_l \in \Omega} h_l(s_{ij})$ 表示语句 s_{ij} 对应长度为 n 的单词计数统计。 $|I|$ 表示图像数量。 $\sum_{l_p \in I} \min(1, \sum_q h_k(s_{pq}))$ 表示词组 k 对应数据集中被描述图像的

数量。

$$g_k(s_{ij}) = \frac{h_k(s_{ij})}{\sum_{w_l \in \Omega} h_l(s_{ij})} \log \left(\frac{|I|}{\sum_{l_p \in I} \min(1, \sum_q h_k(s_{pq}))} \right)$$

$$\text{CIDEr}_n(c_i, S_i) = \frac{1}{m} \sum_j \frac{g^n(c_i) \cdot g^n(s_{ij})}{\|g^n(c_i)\| \|g^n(s_{ij})\|}$$

$$\text{CIDEr}(c_i, S_i) = \frac{1}{N} \sum_{n=1}^N \text{CIDEr}_n(c_i, S_i)$$

NIC 算法在 2015 年 MSCOCO 图像摘要算法竞赛中的相关结果

在 2015 年春季举办的 MSCOCO 图像摘要算法竞赛中, 竞赛主办方采用 CIDEr 作为算法的评价标准。从 2015 年竞赛 MSCOCO 官方网站公布的非公开数据测试结果来看⁴²。以 CIDEr 方法获取的度量数列为标准, 通过公式 $\rho_{X,Y} = \frac{E[(X-\mu_x)(Y-\mu_y)]}{\sigma_X \sigma_Y}$ 计算这些序列的皮尔森相关值, 可知以上评价方法返回结果是高度相关的。它们只在人工标注结果评价方面存在较大差异。表 6-8 由 NIC 原论文给出。它显示了人工标注结果在不同的评价方法下使用 40 个测试案例, 与 MSCOCO2015 图像摘要竞赛的 15 个参赛算法的排序结果。BLEU-4 方法将人工标注的结果排在了非常靠后 13 的位置。CIDEr 则要好很多, 人工标注的排序结果为 6。人工标注语句在 METEOR 评价方法上取得了最好的结果, 排序为 3。

表 6-8 图像摘要评价方法性能对比

	皮尔森相关值(CIDEr)	人工标注结果评价排序
CIDEr	1.0	6
METEOR	0.98	3
ROUGE	0.91	11
BLEU-4	0.87	13

从表 6-8 可以看出目前的图像摘要评价算法尚不能很好地反映图像摘要算法的

42 <http://mscoco.org/dataset/#captions-leaderboard>

实际性能。各评价算法都存在较大的相关性，且对人工标注语句的度量存在一定的误差。语句相似性标准方法的提升对图像摘要问题研究会有本质上的帮助。在 2015 年竞赛时 NIC 算法取得了第一。当时 CIDEr 评价标准的排序结果如表 6-9 所示。

表 6-9 2015MSCOCO 竞赛算法评价前五的结果

	CIDEr	METEOR	ROUGE	BLEU-4	排序
早前版本的 NIC 算法	0.943	0.254	0.53	0.309	1
其他算法名称和相关资料参见 MSCOCO 官方网站公布的测试结果 42	0.931	0.248	0.526	0.308	2
	0.917	0.242	0.521	0.299	3
	0.912	0.247	0.519	0.291	4
	0.886	0.238	0.524	0.302	5
人工标注	0.854	0.252	0.484	0.217	8

NIC 算法含有样本输入、图像特征向量化、单词向量化、LSTM 网络建模等多个模块。它们都对最终的算法性能起到关键作用。实际上，前文详细介绍的 NIC 算法已经进行了优化，各模块的改进点如下：

- 在 ImageNet 的 5 备选分类问题中 Inception_v3 神经网络模型和 GoogLeNet 神经网络模型分类精度分别对应 6.67% 和 4.8%。在图像摘要问题中 Inception_v3 网络同样表现出更好的性能。图像特征向量化方面使用 Inception_v3 模型替换 GoogLeNet 模型可提升图像的特征概括能力，对应的 BLUE-4 的提升幅度约为 2%。
- 优化搜索算法从全局出发从可能的预测序列中挑选出最优的预测结果。与采样方式或贪心方式相比较，使用优化搜索方法输出预测摘要信息能够获取更好结果，对应的 BLUE-4 的提升幅度约为 2%。
- 在完成约 1000000 此初始迭代训练后，再使用 2000000 次迭代允许训练优化 Inception_v3 的网络模型参数可是算法性能得到进一步提升。第二次训练是在第一次基础上进行的，因学习步长是根据指数函数衰减的，二次训练的学习步长远比初始时小很多。二次训练称为微调训练。微调后算法整体性能得到进一步提升。使用二次训练对图像特征向量化模型进行微调。微调后可获取更小的损失函数，对应 BLUE-4 的提升幅度约为 1.5%。
- 与语句的损失函数设计相比，基于单词的损失函数设计能够更好地表征训练过

程与预测过程。此时, BLUE-4 的提升幅度约为 1%。

- 使用 5 个经过 1 次初始训练和 10 个经过 2 次微调训练生成的预测模型分别产生图像摘要结果,并根据置信度挑选出最优结果。模型集成可令 BLUE-4 提升 1.5%。

NIC 算法改进前后在公开的 MSCOCO 数据上的测试结果对比如表 6-10 所示。

表 6-10 NIC 改进后的精度提升对比

	BLEU-4	METEOR	CIDER
初始版本的 NIC 算法	27.7	23.7	85.5
改进后的 NIC 算法	32.1	25.7	99.8

图 6-11 显示的是 NIC 算法按照上表进行改进前后的输出对比例图。测试时随机挑选了 20 张图像进行测试。通过人工比较发现 19 张图输出结果都有所改善, 仅 1 张图输出语句描述效果变差。这进一步表明改进后算法整体性能的提升。



图 6-11 实施模块优化前后 NIC 算法输出结果对比图

6.3 小结

图像处理和自然语言处理在原先似乎一直是并无关联的两个领域,但随着深度学习研究的不断发展,目前已经有多种算法模型将图像的 CNN 和用于处理文本的 LSTM 进行了融合,本章介绍的 ReInspect 和 NIC 就是其中典型的代表。

通过本章的介绍,可以看到使用 TensorFlow 实现各种 CNN+LSTM 的模型的过

程并不复杂，应用也可以快速搭建。在 TensorFlow 的基础上，探索新的领域和尝试新的应用都将充满着无限可能。

6.4 参考资料

- [1] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, Towards Real-Time Object Detection with Region Proposal Networks, arXiv preprint arXiv:1506.01497, 2015
- [2] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun. OverFeat: Integrated recognition, localization and detection using convolutional networks. In ICLR'14
- [3] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In CVPR'14
- [4] S. Zhang, R. Benenson, and B. Schiele. Filtered channel features for pedestrian detection. In CVPR, 2015
- [5] J.R.R. Uijlings, K.E.A. van de Sande, T. Gevers, and A.W.M. Smeulders. Selective search for object recognition. International Journal of Computer Vision, 2013
- [6] Christian Szegedy, Scott Reed, Dumitru Erhan, and Dragomir Anguelov. Scalable, high-quality object detection. CoRR, abs/1412.1441, 2014
- [7] Russell Stewart, Mykhaylo Andriluka, End-to-end people detection in crowded scenes, arXiv:1506.04878, 2015.
- [8] Karen Simonyan, Andrew Zisserman, Very Deep Convolutional Networks For Large-Scale Image Recognition, ICLR 2015

- [9] Navneet Dalal and Bill Triggs, Histograms of Oriented Gradients for Human Detection, In Proceedings of IEEE Conference Computer Vision and Pattern Recognition, San Diego, USA, pages 886-893, June 2005
- [10] Felzenszwalb, P. F. and Girshick, R. B. and McAllester, D. and Ramanan, D. Object Detection with Discriminatively Trained Part Based Models. PAMI, vol. 32, no. 9, pp. 1627-1645, September 2010
- [11] M. Andriluka, S. Roth, and B. Schiele. People-tracking-by-detection and people-detection-by-tracking. In CVPR 2008.
- [12] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. International Journal of Computer Vision, 111(1):98-136, January 2015.
- [13] Victor Lempitsky and Andrew Zisserman. Learning to count objects in images. In NIPS'10.
- [14] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In Neural Information Processing Systems (NIPS), 2015.
- [15] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, Fernando Mujica, Adam Coates, and Andrew Y. Ng. An empirical evaluation of deep learning on highway driving. CoRR, abs/1504.01716, 2015.
- [16] S. Tang, M. Andriluka, A. Milan, K. Schindler, S. Roth, and B. Schiele. Learning people detectors for tracking in crowded scenes. In ICCV'13.
- [17] Christian Wojek, Stefan Walk, and Bernt Schiele. Multi-cue on-board pedestrian detection. In CVPR 2009.

- [18] Oriol Vinyals, Alexander Toshev, Samy Bengio, Dumitru Erhan. Show and Tell: Lessons learned from the 2015 MSCOCO Image Captioning Challenge. IEEE transactions on pattern analysis and machine intelligence (2016).
- [19] A. Farhadi, M. Hejrati, M. A. Sadeghi, P. Young, C. Rashtchian, J. Hockenmaier, and D. Forsyth, "Every picture tells a story: Generating sentences from images," in ECCV, 2010.
- [20] G. Kulkarni, V. Premraj, S. Dhar, S. Li, Y. Choi, A. C. Berg, and T. L. Berg, "Baby talk: Understanding and generating simple image descriptions," in CVPR, 2011.
- [21] K. Cho, B. van Merriënboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in EMNLP, 2014.
- [22] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," arXiv:1409.0473, 2014.
- [23] Has, im Sak, Andrew Senior, Franc, oise Beaufays, Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling

7

损失函数与优化算法

损失函数与目标函数优化算法是深度学习问题中的两个重要概念。这两者有相当的独立性，并都对训练网络的最终结果起到关键的作用。

损失函数（loss function，也叫做代价函数 cost function）是从统计学的角度衡量解决方案错误程度的量化函数。目前的机器学习算法都基于统计学习理论，其重点在于通过有限的训练数据归纳出总体平均意义下的最优规律，用有限的样本逼近全局最优的概率分布，损失函数的作用就是衡量逼近程度。对于机器学习问题来说，给出量化指标，确定优化求解方向，是解决问题的第一步。从这个意义上说，损失函数的设计体现了人们对待求解问题的理解程度。设计优秀的损失函数，不但为问题定义了一个明确的量化指标，还能够提高训练优化速度，因此是机器学习领域非常重要的一个部分。TensorFlow 内置了多种常用损失函数的实现，尤其是，针对大数据训练，TensorFlow 提供了类别采样（Candidate Sampling）损失函数，对减少训练过程中的内存消耗和计算量有非常大的帮助。

另一方面，目标函数的优化问题也是机器学习中的核心问题之一，历来是学术界关注的焦点。在研究和实际应用中，除了最常用的随机梯度下降（Stochastic Gradient Descent, SGD）以外，还相继提出了动量优化算法（Momentum）、ADAM 算法等，这些算法策略不仅加快了求解速度，还同时降低了超参数（如学习率）对求解过程的

影响,简化了训练过程,可谓是有了长足的进步。需要强调的是,考虑到内存消耗和计算量等因素,在深度学习领域提及的目标函数优化方法一般是指支持数据集分块 (mini-batch),并按照分块单元数据包训练的优化算法。本章会重点对梯度下降算法、RMSProp、ADAM 算法在 TensorFlow 中的实现进行详细介绍。

本章将首先介绍 TensorFlow 中已实现的几个常用目标函数优化策略,强调了整体数据集优化与采样 batch 包数据优化的差异,概述 RMSPropOptimizer、AdamOptimizer 和 GradientDescentOptimizer 三个类对应的目标函数优化算法及其对比。其次会介绍对大数据较为适用的类别采样损失函数的相关概念,重点介绍 `nce_loss` 和 `sampled_softmax_loss` 函数对应的类别采样损失函数的相关概念。

7.1 目标函数优化策略

7.1.1 梯度下降算法

梯度下降算法是一种非常简单的目标函数优化算法。以计算函数 $f(x)$ 的最小值为例,梯度下降算法如算法 7-1 所示。

算法 7-1: 梯度下降算法

步骤 1 计算梯度 $f'(x)$

步骤 2 自变量 x 向梯度反方向移动。 $x = x - r \cdot f'(x)$ 。其中, r 为学习率。

步骤 3 循环计算步骤 1 和 2,直至达到最大循环次数或满足 $f(x)$ 收敛条件。则返回 x 作为函数 $f(x)$ 的最小值近似解。

从梯度下降法可以看出学习率 r 的选取对算法的效果起到关键的作用。一方面,如果 r 太大,则可能出现 $f(x)$ 值呈波浪形浮动无法收敛。另一方面,如果 r 太小,则 x 的移动将会非常缓慢,收敛过程将会非常耗时。

TensorFlow 中定义了实现深度神经网络中多种经典类型的网络层,包括卷积层、池化层、激活层、全连接层甚至是循环网络层等。这些网络层对应的输出函数可能是线性的也可能是非线性的。其中,线性函数 $f(x)$ 是指同时满足条件 $f(a+b) = f(a) + f(b)$ 且 $f(k \times a) = k \times f(a)$ 的函数。以矩阵运算 $f(x) = W_f x$ 和 $g(x) = W_g x$ 为例,显然有 $f(g(x)) = W_f W_g x = W x$ 。由此可见,增加线性网络的层数并不能增强神经网络的

函数表示范围。因此,要求在模型中添加非线性网络层构建深度神经网络,以便逼近各种不同的分类或预测模型函数。但为了便于优化模型函数,目前 TensorFlow 所有的网络层的梯度都要求是线性的即 $f'(x)$ 为线性函数。

令训练样本集合构成目标函数 $f(x)$, 其中 x 表示待求解的模型参数。以仅有 1 个输出结果的单个全连接层为例,显然该函数的梯度是线性的。模型参数 x 对应样本 i 的优化目标梯度向量为 $f'^i(x) = f^i(x) - y^i$, 其中 x 表示网络模型参数, y^i 表示目标值向量, $f^i(x)$ 表示样本 i 的单层网络函数输出结果。令 m 表示单次训练分组 batch 数据包中的样本数量,则利用梯度函数的线性特质,可以近似得出模型参数 x 对应 m 个样本的梯度 $\sum_{i=0}^m f'^i(x) = \sum_{i=0}^m (f^i(x) - y^i)$ 。显然,当 batch 大小与数据集样本数量相同时,计算的梯度为训练集全局最优梯度,此时更新模型参数 $x = x - r \cdot \sum_{i=0}^m f'^i(x)$ 往往能得到较理想的结果。

然而,对于数据量较大的复杂模型,训练过程需要反复计算训练集产生的加权梯度。这将导致较长的训练耗时。对于训练样本数量庞大的情况,可以假设对样本集分块后计算的梯度存在高度的相似性,因此可以通过样本集的部分数据计算梯度来近似表示全局最优梯度。使用单个样本计算梯度的极端例子被称为在线学习。与之相比,一般使用几十至几百个样本构成的 batch 数据包方式计算的梯度一般能够得到更好的结果。在使用 batch 数据包训练时,训练样本输入的先后顺序非常重要。如果数据样本不均衡,比如正样本是负样本个数的 1/100,当数据包中连续出现同种类别的样本时,将很可能导致过拟合,部分类别样本的权重更新被吞噬。而对样本序列的随机排序,能够在一定程度上较好的解决样本类别的非均衡问题。此时目标函数优化策略即为随机梯度下降。Tensorflow 中使用 GradientDescentOptimizer 实现了梯度下降算法。而随机梯度下降方法中 batch 数据包中的样本顺序和数据包的大小需要由外部随机函数采样控制。

随机梯度下降方法简单,收敛速度快,是使用较为广泛的优化算法。但它也存在易陷入局部最优导致难以获取最优解的缺点。这有两个方面的原因。一方面是因为网络层模型参数上使用相同的学习率进行梯度更新。另一方面是因为学习率参数完全由人工经验输入。对此,RMSProp、Adam 等算法分别对其做了相应的改进。

7.1.2 RMSProp 优化算法

在随机梯度下降算法中对于所有网络层的梯度分量使用了相同的学习率进行梯度更新,然而这种处理在有些时候是不恰当的。如图 7-1 所示,令图像横轴 w_1 与纵轴 w_2 表示线性模型中两个独立无关的权重分量, $loss$ 表示损失函数值。7-1 左图表示 w_1 、 w_2 构成的等损失函数值曲线图,右图表示 w_1 、 w_2 构成的等错误曲线与对应的损失函数值。从图中我们可以看出 w_1 、 w_2 构成的等损失函数值曲线为同心椭圆。与使用相同的学习率相比,当 w_2 的学习步长大于 w_1 时,算法的收敛速度更快,精度会更高。随机梯度下降无法对不同模型权重设置不同的学习步长。函数优化策略需要根据各梯度分量的实际分布情况,自适应地调整学习步长。在自变量区间内梯度较小且保持一致的情况应采用较大的学习步长。在自变量区间内梯度较大且方向不定的情况应采用较小的学习步长。

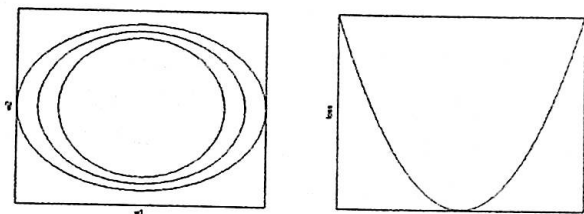


图 7-1 左图表示 w_1 、 w_2 构成的等损失函数值曲线图,右图表示等损失曲线与对应的损失函数值

RMSProp 算法是一种自适应学习率的优化算法,通过统计近似梯度均值的方式来调整学习率。从 TensorFlow 的源代码文件 `rmsprop.py` 中可以看出,当权重梯度均值较大时等比缩小学习率,当权重梯度均值较小时等比放大学习率。RMSPropOptimizer 通过估计训练过程中的梯度方差信息对学习率进行自适应调整。在实现中有两个版本,一个版本是使用衰变因子直接估算梯度均值绝对值,当梯度均值较大时学习率等比下降,反之上升。另一个版本是维护梯度的平均值,并以此预估梯度方差,然后再对学习率进行自适应调整。

算法 7-2: RMSPropOptimizer 使用梯度均值对学习率自适应调整

$$\begin{aligned} \text{mean_square}_t &= \text{decay} \times \text{mean_square}_{t-1} + (1 - \text{decay}) \times g_t^2 \\ \text{mom}_t &= \text{momentum} \times \text{mom}_{t-1} + \text{learning_rate} \times g_t / \sqrt{\text{mean_square}_t + \text{epsilon}} \\ \text{delta}_t &= -\text{mom}_t \end{aligned}$$

算法 7-3: RMSPropOptimizer 使用梯度方差对学习率自适应调整

```

mean_gradt = decay × mean_gradt-1 + (1 - decay) × gt
mean_squaret = decay × mean_squaret-1 + (1 - decay) × gt2
momt = momentum × mom(t-1) +
learning_rate × gt / sqrt(mean_squaret - mean_gradt2 + epsilon)
deltat = -momt

```

算法 7-2 与算法 7-3 中 g_t 表示 t 时刻的梯度, mean_square_t 表示 t 时刻梯度均值平方, decay 表示延迟因子, 用于表示过往数据包统计的梯度对当前梯度计算所起到的作用强度。momentum 表示梯度方向惯性, 起到的作用是即使当前梯度为 0, 亦可使梯度往之前的梯度方向继续调整。 $\text{mean_square}_t - \text{mean_grad}_t^2$ 用于, 估算梯度的方差。在 TensorFlow 的实现中, RMSPropOptimizer 类的构造函数中, 通过参数 centered 控制调用算法 7-2 或算法 7-3。

7.1.3 Adam 优化算法

无论是随机梯度下降还是 RMSProp 优化算法, 学习率 (learning rate) 都是需要人工设置的超参数。尽管 RMSPropOptimizer 在内部实现了对不同模型权重学习率的自适应调整, 其初始基数值仍然是需要人为设置的, 并且学习率的内部调整策略与训练迭代次数无关。按照一般实验观察的结果, 随着迭代次数的增加, 准确率越来越高, 应该缩小更新步长, 否则可能会造成在极小值附近来回震荡。若希望随着迭代次数的增加等比减少学习率基数, 可以根据如下公式进行动态调整。

$$\text{learning_rate}_t = \text{learning_rate}_0 \times \alpha^{\max(0, \frac{\text{iter_num}}{\text{learning_rate_step}} - \text{offset})}$$

Adam 算法与其他优化策略的最大不同在于, Adam 算法使用迭代次数作为参数对梯度均值和梯度均值方差进行了矫正⁴³, 其算法过程如算法 7-4 所示。

⁴³ 算法的完整描述可参见本章参考资料[2]。

算法 7-4: AdamOptimizer 函数优化算法。 g_t^2 表示向量元素的平方。tensorflow 中默认参数为 $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ 和 $\epsilon = 10^{-8}$ 。算法中所有的向量操作都是基于元素操作的。 β_1^t 和 β_2^t 分别表示 β_1 和 β_2 的 t 次方。

令 α 表示学习率基数。

令 β_1 和 β_2 表示两个延迟因子的基数

令 $f(\theta)$ 表示目标函数, 需求解 $\min(f(\theta))$ 时的参数 θ

初始化:

令 θ_0 为均值 0 方差较小的随机向量, $m_0 = 0$, $v_0 = 0$, $t = 0$

while θ_t 没有满足收敛条件时

$t = t + 1$ (更新迭代次数)

$g_t = \nabla_{\theta} f_t(\theta_{t-1})$ (获得目标函数在参数 θ_{t-1} 时的梯度)

$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (估算梯度均值)

$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (估算梯度平方均值)

$\bar{m}_t = m_t / (1 - \beta_1^t)$ (考虑迭代次数对梯度均值进行矫正)

$\bar{v}_t = v_t / (1 - \beta_2^t)$ (考虑迭代次数对梯度平方均值进行矫正)

$\theta_t = \theta_{t-1} - \alpha \cdot \bar{m}_t / (\sqrt{\bar{v}_t} + \epsilon)$

end while

返回 θ_t

相较于 RMSProp 算法, Adam 算法使用迭代次数与延迟因子对梯度均值和梯度平方均值进行了矫正。也正因为如此, Adam 算法对梯度变化的预测可能更加精准。一些实验表明其实验结果要优于 RMSProp。但这并不是绝对的, 最终的训练效果还是要受到训练数据、网络结构、损失函数以及函数优化策略等因素的共同影响。

7.1.4 目标函数优化算法小结

总结来说, GradientDescentOptimizer 实现简单, 梯度更新方向仅由样本特征线性加权获得, 所有特征权重的学习率相同。RMSPropOptimizer 根据学习过程中的参数均值或方差自适应调整每一个模型参数的权重学习率。在 RMSPropOptimizer 的基础上 AdamOptimizer 使用迭代次数对梯度均值和方差进行校正。RMSPropOptimizer 与 AdamOptimizer 的共同点是自动调整学习率。当梯度均值或方差较小时, RMSPropOptimizer 与 AdamOptimizer 会自动增加学习率。无论是哪种函数优化策略, 随着训练次数的增加, 使用小的学习率调整模型参数是十分有必要的。

7.2 类别采样 (Candidate Sampling) 损失函数

根据模型的定义, 损失函数被分为两种: 一种是置信数值的概率转换函数softmax, 另一种是基于概率密度函数的损失函数logistic⁴⁴。本节将重点介绍这两种类别采样 (Candidate Sampling) 损失函数, 并以单词向量化 (word2vec) 为例对基于类别采样的损失函数的优点进行了说明。另外, 本节还对负样本估计类别采样 (Negative Sampling) 损失函数、类别采样logistic (Logistic Sampling) 损失函数进行概括说明。

首先, 问题可以抽象为: 假设一个预测分类问题, 每个训练样本由 (x_i, T_i) 组成。其中 x_i 表示样本 i 的上下文相关特征, T_i 是一个比整个样本类别集合 T 小得多的类别样本子集。

这时, 我们希望学习一个稳定的预测函数 $f(x, y)$, 并由它来估计样本上下文 x 和预测目标 y 的关系。例如, 以概率的形式表示样本上下文和预测结果。

当训练使用所有的数据样本类别更新模型时, 我们在计算softmax或者logistic等操作时需要加载并计算每个训练样例 $f(x, y)$ 对应每个类 $y \in L$ 的预测概率。当类别数 $|L|$ 较大时, 这一操作的内存和计算开销都会非常大。

类别采样技术是针对这一问题的常用技术方法。在构建训练任务时, 对每个训练样例 (x_i, T_i) , 在一个类别子集 C_i 构成的数据包 (batch) 中对函数 $f(x, y)$ 进行评估优化。通常, 类别子集 C_i 由目标类别集合 T_i 和从整体类别集合中随机选择的类别子集 $S_i \subset L$ 构成, 如下所示。

$$C_i = T_i \cup S_i$$

负样本类别集合 S_i 的选择方式可以与上下文特征 x_i 或目标类别 T_i 相关。神经网络模型在训练过程中通过评估预测函数 $f(x, y)$ 在训练数据包中产生的损失函数值来计算反向传播梯度并对模型进行更新。

⁴⁴ Logistic可以理解为机器学习中的模型等价于各目标样本概率密度的分布函数。训练过程只是求解模型的参数。在实际问题中选择与问题样本分布一致的模型结构能够很好地简化训练过程。模型结构过于复杂时会产生大量的冗余参数且训练耗时长。若模型结构过于简单, 则样本可能无法获得良好的收敛。

目前 TensorFlow 中含有的类别采样损失函数主要包括 softmax 和 logistic 两大类型, 如表 7-1 所示, 其中相关概念和函数定义如下:

- softmax 函数常用于单分类问题。这种情况下样本 (x_i, T_i) 只可能含有一个类别标签。logistic 是指模型, 是以概率密度函数的形式表现的。与 softmax 函数不同, logistic 形式的损失函数可用于度量多类问题的损失函数。这种情况下样本 (x_i, T_i) 可以有多个类别标签。
- $Q(y|x)$ 表示类别集合 C_i 中具体采样类别的概率密度函数。其中, x 表示采样函数的上下文相关特征向量, y 表示采样数据包的类别集合 C_i 中单个或多个类别。
- $K(x)$ 表示仅与上下文相关特征向量 x 相关的任何一个可能函数。它与类标签 y 是无关的, 即 $K(x)$ 与任一分类权重和偏移都是无关的。它用来表示分类结果值的基数。分类结果的概率极大值的最终输出会用到归一化操作, 而 $K(x)$ 不会改变最后的判断结果。仅当 $K(x)$ 远大于 $\ln(P(y|x))$ 时, 各类的分类判定概率会趋于均衡。
- logistic 损失函数如下所示。 $G(x_i, y_i)$ 表示样本特征 x_i 对应类标签向量 y_i 的概率置信度相关向量。向量 y_i 由 0、1 构成, 1 表示属于该类别, 0 表示不属于该类别。显然, 向量 y_i 中元素 1 对应的 $G(x_i, y_i)$ 值越大, 损失函数越小。而元素 0 对应的 $G(x_i, y_i)$ 值越小, 损失函数越小。对于 logistic 损失函数, 相同数据集使用全类别集合和子类别采样集合训练时, 向量 $G(x_i, y_i)$ 的长度是不同的。

$$\text{loss} = \sum_i (y_i \times (-\ln(G(x_i, y_i))) + (1 - y_i) \times (-\ln(1 - G(x_i, y_i))))$$

- softmax 的损失函数如下所示。其中 t_i 表示样本 i 的类别标签序号。置信度相关向量 $G(x, y)$ 在目标类别 t_i 的分量值占比越大, 进行 softmax 指数函数归一化时的比例值越小, 则最终的损失值越小。对于 softmax 损失函数, 相同数据集使用全类别集合和子类别采样集合训练时, 向量 $G(x, y)$ 的长度是不同的。

$$\begin{aligned} \text{loss} &= \sum_i (-G(x_i, y)[t_i] + \ln(\sum_{y \in \text{POS}_i \cup \text{NEG}_i} \exp(G(x_i, y)))) \\ &= \ln \frac{\sum_{y \in \text{POS}_i \cup \text{NEG}_i} \exp(G(x_i, y))}{\exp(G(x_i, y)[t_i])} \end{aligned}$$

- 噪声对比估计中的正样本集合 T_i 可表示多个归属类标签。在这种情况下, $P(y|x)$ 表示 y 为目标类别集合 T_i 各元素的所属概率。与此类似, 噪声对比估计、负样本估计和logistic估计中的负类别集合 S_i 也表示为多个归属类标签集合。此时, $Q(y|x)$ 表示 y 为负类别集合 S_i 各类别的所属概率。
- 通过损失函数计算反向传播的梯度时, 全类别集合训练对所有类别 L 的对应分类权重都进行更新。而类别子集采样包则只对指定的类别子集权重进行更新。两种方式都使用了数据包的形式进行反向传播的梯度计算, 因类别采样形式不同, 损失函数的计算是不同的。全类别集合训练是类别子集采样训练的特例。

表 7-1 TensorFlow 类别采样损失函数

	Tensorflow 函数	正/负样本的类别集合 POSi、NEGi	损失函数的输入向量 $G(x, y)$	目标函数向量 $f(x, y)$
softmax 全类别集合损失函数	softmax_loss	正: T_i 单类标签 负: $(L - T_i)$	$f(x, y)$	$\ln(P(y x)) + K(x)$
softmax 类别采样损失函数	sampled_softmax_loss	正: T_i 单类标签 负: $(S_i - T_i)$	$f(x, y) - \ln(Q(y x))$	$\ln(P(y x)) + K(x)$
噪声对比估计类别采样损失函数 Noise Contrastive Estimation (NCE)	nce_loss	正: T_i 多类标签 负: S_i	$f(x, y) - \ln(Q(y x))$	$\ln(P(y x))$
负样本估计类别采样损失函数 Negative Sampling	nce_loss	正: T_i 多类标签 负: S_i	$f(x, y)$	$\ln(\frac{P(y x)}{Q(y x)})$
类别采样 logistic 损失函数	nce_loss	正: T_i 多类标签 负: $(S_i - T_i)$	$f(x, y) - \log(Q(y x))$	$\ln(\frac{P(y x)}{1 - P(y x)})$
类别全集 logistic 损失函数	nce_loss	正: T_i 多类标签 负: $(L - T_i)$	$f(x, y)$	$\ln(\frac{P(y x)}{1 - P(y x)})$

7.2.1 softmax 类别采样损失函数

分类判定问题是指每一个训练样本 $(x_i, \{t_i\})$ 只有一个单一的类别标签。令

$P(y|x) \in (-\infty, +\infty)$ 表示上下文相关特征 x 对应于目标类别 y 的可信度值。**softmax** 函数是指从类别可信度值输出结果中挑选出可信度最大的类别作为输出类别,并将可信度归一化值近似地认定为输出类别的概率。

分类问题的训练目标函数 $f(x, y)$ 是指:在给定上下文相关特征 x 时,通过反向梯度更新分类权重,令目标函数 $f(x, y)$ 取得最小值,并尽可能地使对应类别的可信度对数值 $\ln(P(y|x))$ 取得一个较大值。通过可信度对数值和特征向量相关函数 $K(x)$ 来间接反映目标类别的所属概率。

这里使用对数是为了使概率计算中与类别参数无关项以相加的形式呈现。如下公式所示。

$$F(x, y) \leftarrow \ln(P(y|x)) + K(x)$$

若对分组数据集的所有类别进行 **softmax** 方式的训练,则对于每一个训练样本,需要计算对整体类别集合中每一种类别 y 的目标函数值 $f(x, y)$,并以此来计算反向传播梯度。当整体类别集合非常大时,这种计算策略的计算量会非常大。

使用分组数据类别子集采样包进行 **softmax** 方式的训练时,对于每一组数据通过采样函数 $Q(y|x)$,我们从全类别集合中挑选出一个类别子集 S_i 。每一个标签类别属于集合 S_i 的概率都是独立无关的。对于采样集合 C_i 构成的负样本标签集合的概率如下公式所示。

$$P(S_i = S|x_i) = \prod_{y \in S} Q(y|x) \prod_{y \in (L-S)} (1 - Q(y|x))$$

负类别集合选定后,添加目标类别样本 $\{t_i\}$,重定义训练数据分组的类别集合 $C_i = S_i \cup \{t_i\}$ 。此时,目标函数的训练更新只对类别子集 C_i 所对应的分类参数起作用。

给定一个数据分组的采样类别集合 C_i 和任一样本特征向量 x_i ,对于每一个类别 $y \in C_i$,希望能够计算目标类别的后验概率 $P(t_i = y|x, C_i)$ 。

应用贝叶斯后验概率公式可以展开如下公式:

$$\begin{aligned}
 P(t_i = y | x_i, C_i) &= \frac{P(t_i = y, C_i | x_i)}{P(C_i | x_i)} = \frac{P(t_i = y | x_i) P(C_i | t_i = y, x_i)}{P(C_i | x_i)} \\
 &= \frac{P(y | x_i) P(C_i | t_i = y, x_i)}{P(C_i | x_i)}
 \end{aligned}$$

需要注意的是，采样类别集合 S_i 不包含目标类别集合， S_i 包含采样类别集合 C_i 除目标类别 t_i 的所有元素。 S_i 不会包含不属于 C_i 的元素。使用这一条件可对 $P(C_i | t_i = y, x_i)$ 进行展开如下公式：

$$\begin{aligned}
 P(t_i = y | x_i, C_i) &= \frac{P(y | x_i) P(C_i | t_i = y, x_i)}{P(C_i | x_i)} \\
 &= \frac{P(y | x_i) \prod_{y' \in C_i - \{y\}} Q(y' | x) \prod_{y' \in L - C_i} (1 - Q(y' | x))}{P(C_i | x_i)} \\
 &= \frac{P(y | x_i) \prod_{y' \in C_i} Q(y' | x) \prod_{y' \in L - C_i} (1 - Q(y' | x))}{Q(y | x_i) P(C_i | x_i)} \\
 &= \frac{P(y | x_i)}{Q(y | x_i)} / K(x_i, C_i)
 \end{aligned}$$

对以上公式两边取对数于是有：

$$\ln(P(t_i = y | x_i, C_i)) = \ln(P(y | x_i)) - \ln(Q(y | x_i)) + K'(x_i, C_i)$$

使用分组数据子类别集合训练时，直接使用重定义类别子集，反向传播会使目标函数 $f(x, y)$ 尽可能地“逼近类别子集中期望值” $\ln(P(y | x_i))$ 。由以上公式可知，考虑类别子集 C_i 的采样概率即可完成类别子集与全集训练期望的转换。

灵活性是 TensorFlow 的最大魅力之一，使用 TensorFlow 能够很方便地模拟一个类别数量过万的无监督问题。这里将使用一个较小的文本数据集合来实现单词向量化的训练，并以此为例来说明 `sampled_softmax_loss` 函数的使用。这里使用的单词向量化为优达学城⁴⁵所提供的例子，代码为 TensorFlow 0.12 源码中的 `tensorflow/examples/`

45 优达学城 (Udacity) 是著名的在线 IT 培训机构，它与 Google 合作推出了基于 TensorFlow 的深度学习培训教程，以英文为主。

udacity 目录中的 5_word2vec.ipynb 文件。该程序的功能是对本文文件中出现的英文单词进行向量化表示。要求意思相似的单词向量化表示越相似。因文本文件没有类别标签,为简化问题,假设单词前后上下文越相似则单词含义越相似。基于这一假设即可从文本文件中抽象出一个类别过万的训练集。为便于理解,这一节将对该例子进行展开论述。

首先导入各 Python 的依赖模块。其中 collections 模块中含有集合运算。Math 模块中含有各种数学运算。Numpy 模块可以在 Python 中很方便地使用矩阵。os 为操作系统相关模块。Random 为随机数模块。Tensorflow 为训练模块。Zipfile 为解压模块。Matplotlib 为画图模块。Six.moves 中的 range 为常用幅度区间模块。Six.moves.urllib.request 中的 urlretrieve 为 url 网络文件下载模块。Sklearn.manifold 中的 TSNE 为一个数据降维模块。TSNE 内部含有多种数据降维算法。本文使用 PCA 算法对向量化后的文本进行二维化处理以便于显示。

```
# These are all the modules we'll be using later. Make sure you can
import them
# before proceeding further.
%matplotlib inline
from __future__ import print_function
import collections
import math
import numpy as np
import os
import random
import tensorflow as tf
import zipfile
from matplotlib import pylab
from six.moves import range
from six.moves.urllib.request import urlretrieve
from sklearn.manifold import TSNE
```

函数 maybe_download 从网址 <http://mattmahoney.net/dc/text8.zip> 处下载文本文件并验证文件的大小以确定下载成功。随后使用 zipfile 模块将下载的压缩文件解压。最后使用 tf.compat.as_str 函数将文件以文本形式读入,并使用 split 函数将字符串分割为单词。

```
url = 'http://mattmahoney.net/dc/'
def maybe_download(filename, expected_bytes):
```

```

"""Download a file if not present, and make sure it's the right
size."""
if not os.path.exists(filename):
    filename, _ = urlretrieve(url + filename, filename)
statinfo = os.stat(filename)
if statinfo.st_size == expected_bytes:
    print('Found and verified %s' % filename)
else:
    print(statinfo.st_size)
    raise Exception(
        'Failed to verify ' + filename + '. Can you get to it with a
browser?')
    return filename

filename = maybe_download('text8.zip', 31344016)

def read_data(filename):
    """Extract the first file enclosed in a zip file as a list of
words"""
    with zipfile.ZipFile(filename) as f:
        data = tf.compat.as_str(f.read(f.namelist()[0])).split()
    return data

words = read_data(filename)
print('Data size %d' % len(words))
Data size 17005207

```

从上述运行结果可以看出 text8.zip 压缩后的大小为 30MB 左右。解压后的文本文件长度为 17005207 个单词。为了便于演示减少计算，这里限制单词的种类上线为 50000 种。通过统计词频，可以挑选出词频出现最高的 49999 种类别进行单词向量化，将其他单词统一归类为未知类别 'UNK'。words 对象中保存的原始文本序列构成单词列表。值得注意的是，默认情况 sampled_softmax_loss 函数对类别子集采用对数均匀采样。类别标签应与采样概率值保持一致，即任一对类别序号 $a < b$ ，训练样本集合中类别为 a 的数量应大于 b 。类别集合的对数均匀采样与特征向量无关，它只是一种概括非均衡数据的类别集合采样方式。非均衡数据 (imbalance data) 是指各类别的样本数量差别较大的数据集，例如正样本数量是负样本数量的 20 倍。因为损失函数统计的是整个样本集信息，非均衡导致的最大的问题是样本多的类别会把样本少的类别“吃掉”。在 20 倍数量的差距下 2 分类问题中，负样本的预测即使全错，整体准确率

依然可以高达 95%以上。

当运行完毕函数 `build_dataset` 后，可以获得 `data`、`count`、`dictionary`、`reverse_dictionary` 四个对象。其中 `data` 是一个一元数组列表，里面对应保存的是原始文本序列中的单词在字典 `dictionary` 中的序号。函数 `collections.Counter.most_common` 能对原始文本列表 `count` 进行词频统计。返回的词频中最高的 49999 个单词名称集合和其出现的次数构成二元组列表。字典 `dictionary` 保存的是单词名称和单词频次的排序序号。`reverse_dictionary` 保存的是单词词频的排序序号和对应的单词名称。根据样本数量进行排序并赋予类别标签是按照默认参数调用 `sampled_softmax_loss` 函数的必要步骤。关于这一要求将在后文说明。

```
vocabulary_size = 50000

def build_dataset(words):
    count = [['UNK', -1]]
    count.extend(collections.Counter(words).most_common(vocabulary_size - 1))
    dictionary = dict()
    for word, _ in count:
        dictionary[word] = len(dictionary)
    data = list()
    unk_count = 0
    for word in words:
        if word in dictionary:
            index = dictionary[word]
        else:
            index = 0 # dictionary['UNK']
            unk_count = unk_count + 1
    data.append(index)
    count[0][1] = unk_count
    reverse_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
    return data, count, dictionary, reverse_dictionary

data, count, dictionary, reverse_dictionary = build_dataset(words)
print('Most common words (+UNK)', count[:5])
print('Sample data', data[:10])
del words # Hint to reduce memory.

Most common words (+UNK) [['UNK', 418391], ('the', 1061396), ('of',
```

```
593677), ('and', 416629), ('one', 411764)]
```

```
Sample data [5243, 3083, 12, 6, 195, 2, 3136, 46, 59, 156]
```

通过输出函数可以看出文本集合的单词类别共计 $49999 + 41391 = 91390$ 个。词频最高的单词是 "the"，共计 1061396 次，其次是 "of" 共计 593677 次。"and" 和 "one" 分别出现 416629 和 411764 次为词频出现最高的 3、4 单词。原始文本的前 10 个单词根据词频排序后，在字典中的序号分别是 5243、3083、12、6、195、2、3136、46、59、156。从输出结果可以看出数据处理的正确性。下面将说明数据分组函数及其对应的单次子类别采样训练。

```
data_index = 0
```

```
def generate_batch(batch_size, num_skips, skip_window):
    global data_index
    assert batch_size % num_skips == 0
    assert num_skips <= 2 * skip_window
    batch = np.ndarray(shape=(batch_size), dtype=np.int32)
    labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
    span = 2 * skip_window + 1 # [ skip_window target skip_window ]
    buffer = collections.deque(maxlen=span)
    for _ in range(span):
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    for i in range(batch_size // num_skips):
        target = skip_window # target label at the center of the buffer
        targets_to_avoid = [ skip_window ]
        for j in range(num_skips):
            while target in targets_to_avoid:
                target = random.randint(0, span - 1)
            targets_to_avoid.append(target)
        batch[i * num_skips + j] = buffer[skip_window]
        labels[i * num_skips + j, 0] = buffer[target]
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    return batch, labels

print('data:', [reverse_dictionary[di] for di in data[:8]])

for num_skips, skip_window in [(2, 1), (4, 2)]:
    data_index = 0
```

```

    batch, labels = generate_batch(batch_size=8, num_skips=num_
skips, skip_window=skip_window)
    print('\nwith num_skips = %d and skip_window = %d:' % (num_skips,
skip_window))
    print('    batch:', [reverse_dictionary[bi] for bi in batch])
    print('    labels:', [reverse_dictionary[li] for li in
labels.reshape(8)])

    data: ['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse',
'first']

    with num_skips = 2 and skip_window = 1:
        batch: ['originated', 'originated', 'as', 'as', 'a', 'a', 'term',
'term']
        labels: ['as', 'anarchism', 'a', 'originated', 'term', 'as', 'a',
'of']

    with num_skips = 4 and skip_window = 2:
        batch: ['as', 'as', 'as', 'as', 'a', 'a', 'a', 'a']
        labels: ['anarchism', 'originated', 'term', 'a', 'as', 'of',
'originated', 'term']

```

`generate_batch` 函数负责从原始文本序列中抽取训练数据包。它含有三个参数：`batch_size`、`num_skips`、`skip_windows`。其中 `batch_size` 表示 `batch` 数据包的大小。`num_skips` 表示从原始文本序列中的扫描窗口内采样的样本个数。`skip_windows` 表示扫描窗口的半径含有多少个单词。`data_index` 表示文本序列当前单词索引。`buffer` 表示一个先进先出的队列。在函数开始时 `data_index` 为 0，`buffer` 中被填入了 $\text{skip_windows} \times 2 + 1$ 个单词形成扫描窗口。在每一个生成的扫描窗口中采样 `num_skips` 次。每次采样使用扫描窗口的中心位置作为特征单词。窗口内随机选择不重复的样本位置作为类别标签单词。扫描窗口采样完毕后，更新 `data_index` 值，向队列 `buffer` 添加新的单词，刷新扫描窗口位置，继续采样。值得注意的是 `batch_size` 必须能够被 `num_skips` 整除，以确保在整数次窗口采样中能够获取 `batch` 训练包的样本。`num_skips` 必须小于等于 $\text{skip_windows} \times 2$ ，确保每一个样本最多仅被采样一次。最后令 `batch_size` 为 8，打印了原始文本序列的前 8 个单词，并分别在 `num_skips=2`、`skip_windows=1` 和 `num_skips=4`、`skip_windows=2` 的情况下显示生成的数据包。

由此可知，此时 `batch` 数据包对应的生成函数是与上下文相关的。每个单词词频

差异较大, 整个样本集合是一个类别数量庞大的非均衡数据集。直接使用类别全集对采样包进行训练是十分耗时的。每次梯度更新都至少需要对 50000 个类别的分类权重向量进行更新。当 batch 数据包样本远小于类别数量时, 此时计算的分类梯度误差较大。若使用 `sampled_softmax_loss` 函数, 则分类权重参数的更新只在采样类别子集 C_i 中发生, 分类问题的类别数量被限制在与样本数据包相差不大的范围内。在本例的训练过程中 batch 数据包大小为 128, 采样的负样本类别数量为 64。即单次数据包的训练, 因重定义了类别集合, 只计算更新 192 个类别的分类权重参数。考虑采样函数中的 $-\ln(Q(y|x_i))$, 则可将类别子集期望输出转换为类别全集的期望输出。

如公式 $C_i = T_i \cup S_i$ 所示, 这里的目标类别集合 T_i 由 batch 样本包中的目标类别构成, 而负类别集合 S_i 是通过对数均匀分布采样函数选取的。对数均匀分布将在下一节中详细介绍。TensorFlow 中提供了多种类别子集的采样函数, 可作为参数传入函数 `sampled_softmax_loss` 中。

需要强调的是, `sampled_softmax_loss` 与 `softmax` 函数的不同是, `softmax` 适用于类别数量较少的单分类问题的损失函数计算, 当样本的类别数量较多时, `softmax` 损失函数更新模型会非常缓慢。`sampled_softmax_loss` 通过计算类别子集损失函数, 对类别数量较大的分类问题进行了加速。

```
batch_size = 128
# 单词向量化的维度。
embedding_size = 128
# 单词序列扫描窗口半径。
skip_window = 1
# 每个扫描窗口的采样次数。
num_skips = 2
# 从词频最高的前 valid_window 个单词中随机选择 valid_size 个, 作为验证集查看向量化后最邻近向量的单词名称。
# 验证集的大小。
valid_size = 5
# 固定验证集的位置。
valid_window = 103
valid_examples = np.array(range(valid_window, valid_window +
valid_size))
# tensorflow.nn.sampled_softmax_loss 函数中负样本类别集合的大小
num_sampled = 64
```

定义了相关网络结构参数后,就可以使用它们生成具体的网络。为简化问题,这里仅使用一层线性网络进行模型训练。因为 TensorFlow 使用张量进行损失函数的权重更新,这种更新策略可以发生在输入层。只要输入层的定义为 `tf.variable`,就能够对其进行更新。在源码中,使用的是 `AdagradOptimizer` 优化器⁴⁶。它与 `AdamOptimizer` 类似,是一种根据梯度信息自动更新学习率的函数优化策略。`AdagradOptimizer` 的梯度更新策略较复杂,且需要保存过往历史梯度信息,部分实践表明 `AdamOptimizer` 算法优于 `AdagradOptimizer` 算法。

在本例中,因不同的单词类别可能会含有相同的上下文,上下文的相关程度越高,共同出现的频次越高的向量应越相似。初始时,定义一个大小为 50000×128 的张量,服从区间 $[-1,1]$ 的均匀分布,令每一行表示一个向量化后的单词。使用函数 `tf.nn.embedding_lookup` 能够在 TensorFlow 的张量中获取指定行号的子集形成训练采样包。最后定义验证方式,在验证时使用归一化向量的内积表示两单词向量的相似程度。

全类别集的 `softmax` 损失函数可直接使用如下语句完成损失函数的计算。`tf.nn.softmax(tf.matmul(inputs, tf.transpose(weights)) + biases)`。然而,当类别数量非常大,例如 50000 种时,显然按照 `tf.nn.softmax` 的训练是非常耗时的。`sampled_softmax_loss` 可加速完成训练。基于前文介绍的相关概念,TensorFlow 中的函数 `sampled_softmax_loss` 会根据 `batch` 数据包的类别,从类别集合中额外随机选取一定数量的类别作为负样本类别子集。然后,这些负样本类别子集再和采样包样本类别集合组合并训练计算损失函数值。默认时 `sampled_softmax_loss` 函数使用对数均匀分布函数对类别进行采样,这是一种针对非均衡数据的类别采样方法。此时,在调用 `sampled_softmax_loss` 函数前,需要对训练集中的样本数量对类别标签进行排序,将样本量大的类型赋予较小的类别标签。

```
graph = tf.Graph()
```

```
with graph.as_default(), tf.device('/cpu:0'):
```

```
    #输入数据
```

46 `AdagradOptimizer` 内部实现可参阅本本章参考资料[5]。参考资料[2]中的实践表明 `AdamOptimizer` 优化策略性能优于 `AdagradOptimizer`。

```

train_dataset = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

# 需要训练更新的变量
embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
softmax_weights = tf.Variable(
    tf.truncated_normal([vocabulary_size, embedding_size],
                        stddev=1.0 / math.sqrt(embedding_size)))
softmax_biases = tf.Variable(tf.zeros([vocabulary_size]))

# 模型定义
# 使用嵌入式 tensor 子集
embed = tf.nn.embedding_lookup(embeddings, train_dataset)
# 计算 softmax 损失函数, 每次计算在采样包内选取一个类别子集作为负样本类别。
loss = tf.reduce_mean(
    tf.nn.sampled_softmax_loss(softmax_weights, softmax_biases,
                                embed, train_labels, num_sampled,
                                vocabulary_size))

# 函数优化器
# 优化器将会根据损失函数更新 1 层分类器的网络权重和单词的嵌入式 tensor 本身。
这是因为 tensor embeddings 是通过函数 tf.Variable 定义的。优化器将根据损失函数
梯度, 利用反向传播机制更新整个网络中所有 tf.Variable 定义的变量。
optimizer = tf.train.AdamOptimizer().minimize(loss)

# 使用余弦距离计算采样包验证集向量的差异
norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1,
keep_dims=True))
normalized_embeddings = embeddings / norm
valid_embeddings = tf.nn.embedding_lookup(
    normalized_embeddings, valid_dataset)
similarity = tf.matmul(valid_embeddings, tf.transpose(normalized_
embeddings))

在训练时将计算图代入 session。进行必要的张量初始化后, 对原始单词序列进
行 100001 次循环的训练迭代, 平均 2000 次输出计算的损失均值。平均每迭代训练
10000 次, 输出 1 次验证集的验证结果。验证时只在限定的验证集合内部挑选出与验
证目标最接近的 8 个相似单词。

num_steps = 100001

```

```

with tf.Session(graph=graph) as session:
    tf.global_variables_initializer().run()
    print('Initialized')
    average_loss = 0
    for step in range(num_steps):
        batch_data, batch_labels = generate_batch(
            batch_size, num_skips, skip_window)
        feed_dict = {train_dataset : batch_data, train_labels :
batch_labels}
        _, l = session.run([optimizer, loss], feed_dict=feed_dict)
        average_loss += l
        if step % 2000 == 0:
            if step > 0:
                average_loss = average_loss / 2000
                # 每 2000 次采样包训练迭代输出一次损失函数。
            print('Average loss at step %d: %f' % (step, average_loss))
            average_loss = 0
            # 验证过程是耗时的每迭代 500 次进行一次验证约延迟 20%左右
            if step % 10000 == 0:
                sim = similarity.eval()
                for i in range(valid_size):
                    valid_word = reverse_dictionary[valid_examples[i]]
                    top_k = 8 # number of nearest neighbors
                    nearest = (-sim[i, :]).argsort()[1:top_k+1]
                    log = 'Nearest to %s:' % valid_word
                    for k in range(top_k):
                        close_word = reverse_dictionary[nearest[k]]
                        log = '%s %s,' % (log, close_word)
                    print(log)
            final_embeddings = normalized_embeddings.eval()

```

运行上述程序后，模型的初始损失值为 8.4244。经过 100001 次迭代训练后降为 3.4327。初始状态下，单词 english 和 under 的特征向量相似，单词其语义与单词本身没有关系。当完成训练后，english 和 under 对应的特征向量在特征空间中能够较好地找到语义相似词。例如与 english 相似的单词包括 french, german, irish 等，与 under 相似的单词包括 within, following, halts 等。

```

Initialized
Average loss at step 0 : 8.42440223694
Nearest to english: rubicon, krueger, uncontested, meow, cared, who,
heapsort, assistance,

```

Nearest to then: simultaneity, epidemic, mppc, neutron, named, adventurers, greece, synchronization,

Nearest to any: moi, designation, realignment, brixton, musings, ethic, perineum, differed,

Nearest to both: begin, leaderships, kinderhook, negev, somewhat, recursive, switchable, zionist,

Nearest to under: ich, beta, victorians, waldorf, recollections, searchable, ellipsis, biota,

.....

Average loss at step 100000 : 3.43271456218

Nearest to english: french, german, irish, canadian, swedish, italian, arabic, spanish,

Nearest to then: rarely, therefore, thus, finally, xenon, eventually, hermes, renormalization,

Nearest to any: every, each, no, some, the, a, norwood, another,

Nearest to both: all, various, classifying, bled, seventeen, expelled, micas, abuses,

Nearest to under: within, following, halts, through, haploid, in, between, caret,

程序最后从训练的归一化向量集中挑选出 400 个词频最高的单词向量。使用 TSNE 类，用基于主成分分析 PCA 的降维算法对单词向量降为至 2 维，以便于在平面坐标系中显示。关于 TSNE 类的使用可以从 Python 的 sklearn 模块中了解其相关功能和参考文献。

```
num_points = 400
tsne = TSNE(perplexity=30, n_components=2, init='pca',
n_iter=5000)
two_d_embeddings = tsne.fit_transform(final_embeddings[1:num_
points+1, :])
```

最后 plot 函数负责对降维后的向量和单词名称画出散点图。如图 7-1 所示。

```
def plot(embeddings, labels):
    assert embeddings.shape[0] >= len(labels), 'More labels than
embeddings'
    pylab.figure(figsize=(15,15)) # in inches
    for i, label in enumerate(labels):
        x, y = embeddings[i,:]
        pylab.scatter(x, y)
    pylab.annotate(label, xy=(x, y), xytext=(5, 2), textcoords='offset
points',
```


`sampled_softmax_loss`函数, 仅参数 `subtract_log_q=True` 不同。该参数表明损失误差需要减去类别采样的概率值, 以实现全类别集的损失函数转换。在不同的类别采样损失函数中该值的具体含义是不同的。在`sampled_softmax_loss`函数中, 该值必须为 `True`。`_compute_sampled_logits` 函数的完整源码可在 `tensorflow/python/ops/nn_impl.py` 文件查看。

表 7-2 `sampled_softmax_loss` 函数说明

参数名称	说明
<code>weights</code>	Tensor 类型, 尺寸为 <code>[L, dim]</code> 。其中 <code>L</code> 为全类别集合, <code>dim</code> 为线性分类器特征维数。 <code>weights</code> 表示各类别的线性分类器特征权重
<code>biases</code>	Tensor 类型, 尺寸为 <code>[L]</code> 。 <code>biases</code> 表示各类别的线性分类器偏移
<code>inputs</code>	Tensor 类型, 尺寸为 <code>[batch_size, dim]</code> 。其中 <code>batch</code> 表示采样包样本数量。 <code>dim</code> 是样本特征维数。 <code>inputs</code> 表示采样包数据集
<code>labels</code>	Tensor 类型, 数据类型为 <code>int64</code> 。 <code>labels</code> 表示原始数据集的数据类别
<code>num_sampled</code>	<code>int</code> 类型, <code>num_sampled</code> 表示负样本类别的采样数量
<code>num_classes</code>	<code>int</code> 类型, <code>num_classes</code> 表示可能的类型集合
<code>num_true</code>	<code>int</code> 类型, 样本可能含有的类别数量。 <code>softmax</code> 类型损失函数只能为 1
<code>sampled_values</code>	为(“采样集合”, “目标期望率”, “采样期望率”)
<code>remove_accidental_hits</code>	<code>bool</code> 类型, 标记是否当负样本估计类别采样集合中含有目标类别样本时, 将该类别移除
<code>partition_strategy</code>	数据包划分策略。当 <code>weights</code> 表示一个大型 tensor 的划分组合时, <code>partition_strategy</code> 用于表示划分的策略。包括 <code>div</code> 整除连号划分和 <code>mod</code> 余数相等划分两种策略。若 <code>weights</code> 不存在划分, 则该值无意义
<code>name</code>	指明操作名称。 <code>sampled_softmax_loss</code> 操作名称为“ <code>sampled_softmax_loss</code> ”

TensorFlow 中 `_compute_sampled_logits` 函数的代码实现如下所示。该函数需要对输入参数做一些转换, 便于后续的子集转换。首先是判断输入参数 `weights` 并转换为列表形式。其次是将类别标签转换成类型为 64 位整数的 1 维张量。

```
def _compute_sampled_logits(...subtract_log_q=True...):
    # 若输入参数 weights 不为列表类型, 则将其转换为列表类型。
    if not isinstance(weights, list):
        weights = [weights]

    # 函数 ops.name_scope 从列表 weights + [biases, inputs, labels] 反向
    定位到所处的网络图。并用参数 name 或 "compute_sampled_logits" 创建当前网络图栈
```

顶创建一个块，当 name 值为空时使用。可在运行时通过指定操作名称运行操作。在使用 tensorboard 画图时也将呈现该名称。

```
with ops.name_scope(name, "compute_sampled_logits",
                    weights + [biases, inputs, labels]):
    if labels.dtype != dtypes.int64:
        labels = math_ops.cast(labels, dtypes.int64)
    labels_flat = array_ops.reshape(labels, [-1])
```

#若标签类型不为 64 位整形，则对其转换。并将类别标签转换位 1 维数组。

```
if labels.dtype != dtypes.int64:
    labels = math_ops.cast(labels, dtypes.int64)
    labels_flat = array_ops.reshape(labels, [-1])
```

使用默认的对数均匀分布对样本集类别进行采样。这种采样方式是专门针对非均衡数据的大类别集训练的。可从 TensorFlow 源码 range_sampler.cc 中定位到关键的 LogUniformSampler::Sample 函数。

```
int64 LogUniformSampler::Sample(random::SimplePhilox* rnd) const
{
    const int64 value =
        static_cast<int64>(exp(rnd->RandDouble() * log_range_) - 1;
    CHECK_GE(value, 0);
    // Mathematically, value should be <= range_, but might not be due
    to some
    // floating point roundoff, so we mod by range_.
    return value % range_;
}
```

假设整个样本集合有 L 个类别 $\{0, 1, \dots, t, \dots, L-1\}$ ，采样类别为 t ， x 表示 $\text{RandDouble()} * \log_range$ 语句产生的 $[0, \ln(L))$ 区间的均匀分布的随机变量。仅当 $x \in [\ln(t+1), \ln(t+2))$ 时，采样函数获取类别 t 。因此类别 t 的采样命中概率为 $p(t)$ 。可见，当 t 越大时，样本集中的出现频率越低，而作为负样本类别的采样密度函数越低。这种处理方式确保了不同样本类别概率分布的平滑性。

$$p(t) = \frac{\ln(t+2) - \ln(t+1)}{\ln L}$$

负样本类别的采样操作初始值由函数 log_uniform_candidate_sampler 完成，该函数内部核心由 C++ 实现，有兴趣的读者可以在 candidate_sampling_ops.cc 文件中查看。

在源码文件 `range_sampler.cc` 的函数 `RangeSample::SampleBatchGetExpectedCountAvoid` 中可以查看采样类别的期望概率的具体计算实现。

类别子集预测期望 $Q(y|x_i)$ 对应的类别采样概率为 $1 - (1 - p(t))^{\text{num_tries}}$, $p(t)$ 表示类别 t 的采样命中概率。其中 `num_tries` 表示采样次数。 $(1 - p(t))^{\text{num_tries}}$ 表示 `num_tries` 次采样连续不命中的概率值。令 `num_sampled` 表示负样本类别采样的数量。当函数 `sampled_softmax_loss` 作为损失函数时, 因函数 `log_uniform_candidate_sampler` 中的参数 `unique` 为 `True`, 所以负样本类别不可包含重复类别, 对应的采样次数 `num_tries` \geq `num_sampled`。这里并没有排除负样本类别与目标类别重复的情况, 还需要对这种特殊情况进行晒出。

```
#函数 log_uniform_candidate_sampler 实现负样本类别的采样。
# sampled_values 为三元组:
#   sampled 表示负样本类别集合尺寸为 [num_sampled] 的 tensor
#   true_expected_count 表示目标样本对应类别的在负样本类别集合中出现的期望概率。尺寸为 [batch_size, 1] 的 tensor
#   sampled_expected_count 表示负样本类别集合在采样期望概率。其尺寸为 [num_sampled] 的 tensor
if sampled_values is None:
    sampled_values = candidate_sampling_ops.log_uniform_candidate_sampler(
        true_classes=labels,
        num_true=num_true,
        num_sampled=num_sampled,
        unique=True,
        range_max=num_classes)
# 注意: 为了便于 pylint 解析源码结构需要对元组 'sampled_values' 进行拆分
sampled, true_expected_count, sampled_expected_count = sampled_values
```

为了方便 `batch` 数据包的类别采样训练, 需要将目标类别标签展开并和采样的负样本类别合并。`all_ids` 表示整个采样包类别集合。`all_w` 表示采样包内全集类别的分类权重矩阵, `all_b` 表示对应的分类偏移 `tensor`。`true_w` 表示目标类别的分类权重参数, 从 `all_w` 裁剪得出。`true_b` 表示目标类别的分类偏移参数, 从 `all_b` 中裁剪得出。函数 `array_ops.slice` 有 3 个输入参数, 参数 1 是被切分的 `tensor`, 参数 2 是对应被切分 `tensor` 的维数偏移, 参数 3 是切分长度。

```

# labels_flat 是尺寸为 [batch_size * num_true] 的 tensor
# sampled 是尺寸为 [num_sampled] 的 tensor
all_ids = array_ops.concat(0, [labels_flat, sampled])

# weights 是整体数据类别的分类权重矩阵尺寸为 [num_classes, dim]。
bias 是分类位移偏量。
all_w = embedding_ops.embedding_lookup(
    weights, all_ids, partition_strategy=partition_strategy)
all_b = embedding_ops.embedding_lookup(biases, all_ids)
# true_w 尺寸为 [batch_size * num_true, dim]
# true_b 尺寸为 [batch_size * num_true] tensor
true_w = array_ops.slice(
    all_w, [0, 0], array_ops.pack([array_ops.shape(labels_flat)
[0], -1]))
true_b = array_ops.slice(all_b, [0], array_ops.shape(labels_
flat))

```

使用下面的代码计算输入目标类别特征的线性分类判别结果。true_logits 保存最后的计算结果。

```

# inputs shape is [batch_size, dim]
# true_w shape is [batch_size * num_true, dim]
# row_wise_dots is [batch_size, num_true, dim]
dim = array_ops.shape(true_w)[1:2]
new_true_w_shape = array_ops.concat(0, [[-1, num_true], dim])
row_wise_dots = math_ops.mul(
    array_ops.expand_dims(inputs, 1),
    array_ops.reshape(true_w, new_true_w_shape))
# We want the row-wise dot plus biases which yields a
# [batch_size, num_true] tensor of true_logits.
dots_as_matrix = array_ops.reshape(row_wise_dots,
    array_ops.concat(0, [[-1], dim]))
true_logits = array_ops.reshape(_sum_rows(dots_as_matrix), [-1,
num_true])
true_b = array_ops.reshape(true_b, [-1, num_true])
true_logits += true_b

```

从类别全集分类权重 all_w 中挑选出负样本类别子集的分类权重 sampled_w。并用类似的方法提取出分类偏移 sampled_b。最后，计算采样特征对应负样本类别的线性分类器分类置信度，并保存在 sampled_logits 中。每一行表示一个采样样本在负样本类别子集中的分类置信度。

```

# 通过索引方式获取负样本类别分类权重和偏移量
# sampled_w 的尺寸为 [num_sampled, dim]
# sampled_b 的尺寸为 [num_sampled]
sampled_w = array_ops.slice(
    all_w, array_ops.pack([array_ops.shape(labels_flat)[0],
0]), [-1, -1])
sampled_b = array_ops.slice(all_b, array_ops.shape(labels_flat),
[-1])

```

```

# inputs 的尺寸为 [batch_size, dim]
# sampled_w 的尺寸为 [num_sampled, dim]
# sampled_b 的尺寸为 [num_sampled]
# Apply X*W'+B, which yields [batch_size, num_sampled]
sampled_logits = math_ops.matmul(
    inputs, sampled_w, transpose_b=True) + sampled_b

```

sampled_softmax_loss中, 若负采样类别中含有目标类别则应移除。函数 compute_accidental_hits 负责获取目标类别集合和采样类别集合的重叠标签部分。其中 acc_indices 保存重叠类别在采样类别 labels 中的下标位置, acc_ids 标记重叠部分在类别采样集合中的下标位置, acc_weights 尺寸与 acc_indices 和 acc_ids 的尺寸相同, 且其内值为-FLT_MAX。构建稀疏矩阵坐标 sparse_indices 用于标记 sampled_logits 中对应的采样负样本类别与目标类别重叠的位置。最后对 sampled_logits 中的分类置信度进行修正, 使与目标类别重叠的分类置信度为-FLT_MAX。

```

if remove_accidental_hits:
    acc_hits = candidate_sampling_ops.compute_accidental_hits(
        labels, sampled, num_true=num_true)
    acc_indices, acc_ids, acc_weights = acc_hits

    # This is how SparseToDense expects the indices.
    acc_indices_2d = array_ops.reshape(acc_indices, [-1, 1])
    acc_ids_2d_int32 = array_ops.reshape(
        math_ops.cast(acc_ids, dtypes.int32), [-1, 1])
    sparse_indices = array_ops.concat(1, [acc_indices_2d,
acc_ids_2d_int32],
                                "sparse_indices")

    # Create sampled_logits_shape = [batch_size, num_sampled]
    sampled_logits_shape = array_ops.concat(
        0,
        [array_ops.shape(labels)[:1], array_ops.expand_dims(num_
sampled, 0)])

```

```

        if sampled_logits.dtype != acc_weights.dtype:
            acc_weights = math_ops.cast(acc_weights, sampled_logits.
dtype)
        sampled_logits += sparse_ops.sparse_to_dense(
            sparse_indices,
            sampled_logits_shape,
            acc_weights,
            default_value=0.0,
            validate_indices=False)

```

#使目标类别分类置信度与负样本类别分类置信度加上类别采样概率的影响。

```

        if subtract_log_q:
            # Subtract log of Q(l), prior probability that l appears in
sampled.
            true_logits -= math_ops.log(true_expected_count)
            sampled_logits -= math_ops.log(sampled_expected_count)

```

构建采样类别中的分类置信度输出 `out_logits` 和样本标签 `out_labels`。在 `sampled_softmax_loss` 中, `out_logits` 的每一行表示一个特征样本对应的目标类别和负样本类别的置信度输出, 且目标样本的置信度输出在前 `num_true` 列。 `sampled_softmax_loss` 中 `num_true` 为 1。 `out_labels` 的每一行表示特征样本对应的类别标签, 其中只有前 `num_true` 列为 1/`num_true`。

```

        # Construct output logits and labels. The true labels/logits
start at col 0.
        out_logits = array_ops.concat(1, [true_logits, sampled_logits])
        # true_logits is a float tensor, ones_like(true_logits) is a
float tensor
        # of ones. We then divide by num_true to ensure the per-example
labels sum
        # to 1.0, i.e. form a proper probability distribution.
        out_labels = array_ops.concat(1,
                                     [array_ops.ones_like(true_logits) /
num_true,
                                     array_ops.zeros_like(sampled_logits)])

```

至此 `_compute_sampled_logits` 函数执行完毕, 它的返回值 `logits` 和 `labels` 被代入到函数 `nn_ops.softmax_cross_entropy_with_logits` 中按照前文介绍的 softmax 损失函数定义计算并返回一个尺寸为 `batch_size` 的 tensor。

7.2.2 噪声对比估计类别采样损失函数

当训练数据中的每一个样本 (x_i, T_i) 由上下文相关特征 x_i 和多类标签集合 T_i 组成时,一个样本可能含有多个类标记。softmax的损失函数估计方法使用分类置信度函数转换概率的方式来预测单个类标签的分类问题。与此不同,噪声对比估计类别采样方法,将整个网络结构认定为一个概率密度函数。输入的数据包括属于目标类别 T_i 数据和已知概率分布的噪声样本。基于这些假设,噪声对比估计损失函数期望能最大化区分目标类别数据和噪声数据的概率密度函数区分度。通过反向传播梯度训练模型参数来近似计算符合目标样本概率分布的概率密度函数。

噪声对比估计的模型为:定义非归一化概率密度函数 $p_m^0(\cdot; a)$,令

$$\ln(p_m^0(\cdot; \theta)) = \ln(p_m^0(\cdot; a)) + c$$

总存在一个 c 值使得

$$\int_{-\infty}^{+\infty} p_m^0(\cdot; \theta) = 1$$

定义概率密度函数服从 $p_m^0(\cdot; \theta)$ 分布的目标类别数据 $X = (x_1, x_2, \dots, x_T)$ 和某种已知概率密度分布函数的噪声数据 $Y = (y_1, y_2, \dots, y_T)$,则整个样本集 $U = (u_1, u_2, \dots, u_{2T})$ 是由数据集 X 和数据集 Y 所构成。样本 u_t 的数据类别由类别标签 C_t 表示。当且仅当 $C_t = 1$ 时, $u_t \in X$ 。当且仅当 $C_t = 0$ 时, $u_t \in Y$ 。 X 的概率密度函数形式由模型结构定义,模型参数由训练计算得出。关于类别 X 和类别 Y 的概率函数如下公式所示。

$$P(u|C=1; \theta) = p_m(u; \theta)$$

$$P(u|C=0) = p_n(u)$$

总可以假设类别标签的出现概率相同,即 $P(C=0) = P(C=1) = 0.5$ 。于是我们可以获取如下公式所示的后验概率密度函数。

$$P(C=1|u; \theta) = \frac{p_m(u; \theta)}{p_m(u; \theta) + p_n(u)} = h(u; \theta)$$

$$h(u; \theta) = \frac{1}{1 + \exp[-G(u; \theta)]} = r(G(u; \theta))$$

$$P(C = 0|u; \theta) = 1 - h(u; \theta)$$

其中, 令 $r(\cdot)$ 表示指数函数,

$$G(u; \theta) = \ln(p_m(u; \theta)) - \ln(p_n(u))$$

另外, 从以上公式可以看出样本 u 属于类别为“1”的概率服从泊松分布。泊松分布被广泛用于单位时间内时间发生的概率统计。于是, 对于整个样本集信息熵函数有如下公式所示:

$$\begin{aligned} l(\theta) &= \sum_t (C_t \ln(P(C_t = 1|u_t; \theta)) + (1 - C_t) \ln(P(C_t = 0|u_t; \theta))) \\ &= \sum_t (\ln[h(x_t; \theta)] + \ln[1 - h(y_t; \theta)]) \end{aligned}$$

此时的训练问题就是训练参数 θ , 使得以上公式取最大值。即最大化区分样本类别的概率密度分布函数和噪声概率密度分布函数的区分度。对以上公式进一步展开, 以下公式与logistic损失函数的定义是等价的。

$$l(\theta) = \sum_t (\ln[r(\ln p_m(u; \theta) - \ln p_n(u))] + \ln[1 - r(\ln p_m(u; \theta) - \ln p_n(u))])$$

由此可见, 目标函数会与负样本的概率密度函数相关。给定上下文相关特征, 需要估算出对应特征向量的目标类别集合中各类别的概率密度概率值期望

$$P(y|x) = E(T(y)|x)$$

当目标类别集合中仅含有一个类时, 此时的训练输出就仅仅是该类别的概率值。负样本的概率密度函数是已知的或是预先定义好的。

噪声对比估计的特点是使用负样本的概率密度函数与目标概率密度函数进行对比。前文已经提到, 定义好的网络模型就是我们假设的目标概率密度函数 $f(x, y)$ 。而训练过程就是调整模型的内部参数, 使函数 $f(x, y)$ 尽可能地逼近特征向量 x 对应的目标类别 y 的相对对数后验概率对数值, 即

$$f(x, y) \leftarrow \ln(P(y|x)) \leftarrow \ln(p_m(u; \theta))$$

对于训练样本 (x_i, T_i) ，我们对单个的类标签进行拆分合并形成集合，但这里使用多标签组抽样描述是为了不失一般性。例如，多类标签组可以是 $L = \{\{1,2\}, \{3,4,5\}, \{3,4\}\}$ 这种形式，拆分合并后的类别集合是 $L = \{1,2,3,4,5\}$ 。噪声对比估计的噪声类别抽样函数可以与上下文特征 x 相关或无关，但不应与目标类别 T_i 相关。构建一个类别子集包括目标类别集合和负样本类别集合的所有类别。这里允许负样本类别集 S_i 中含有与目标样本 T_i 含有相同的类别标签。

$$C_i = T_i + S_i$$

此时，我们的训练任务就是通过目标函数的概率密度输出和噪声样本的概率密度输出计算损失函数。经过训练，从目标类别和噪声类别中判别出样本的目标类别。对于正样本目标类别 T_i 和采样负样本类别 S_i 都有一个代表性的训练样本实例，即这两个集合的样本上下文应具有一定的区分度。

不失一般性，在负样本概率分布函数未知的情况下，令 $Q(y|x)$ 表示给定上下文相关特征采样类别的概率分布。令噪声类别组集合 S_i 不存在重复的单个类标签则有如下公式：

$$Q(y|x) = E(S(y)|x) = p_n(u)$$

令负样本类别采样概率为噪声概率密度分布有如下公式：

$$\begin{aligned} G(u; \theta) &= \ln(p_m(u; \theta)) - \ln(p_n(u)) = \ln(P(y|x)) \\ -\ln(Q(y|x)) &= G(x, y) = \ln\left(\frac{\text{模型预测概率}}{\text{采样概率}(S)}\right) \end{aligned}$$

通过以上转换可将噪声对比估计变换为softmax类别采样的目标函数的计算形式，即有 $G(x, y) = \ln(P(y|x)) - \ln(Q(y|x))$ 。其中， $P(y|x)$ 是采样类别子集的概率期望输出。在此基础上，引入 $-\ln(Q(y|x))$ 来表示对比噪声的概率密度函数，即实现了数据包类别子集损失函数值至全类别集合的概率的转换。噪声对比估计对模型参数的更新只发生在类别子集内部。

实际上 TensorFlow 内部的nce_loss函数和sampled_softmax_loss函数的实现都采

用了 `_compute_sampled_logits` 函数来计算损失函数的输入参数。因为实际训练样本中并不存在噪声负样本，按照噪声对比估计联合概率信息熵的定义，当 $G(u; \theta)$ 取最大值时信息熵可取得最大值。因此，`nce_loss` 函数与 `sampled_softmax_loss` 函数的不同在于以下两点。

1. `nce_loss` 函数内部调用 `_compute_sampled_logits` 时，输入参数 `remove_accidental_hits` 为 `False`。也就是说，噪声对比估计允许负样本估计类别采样类别中含有目标样本类别。在上一节已经强调，整个模型被认为是区分目标类别概率分布函数和负样本类别概率分布函数的判定函数，即同一上下文特征中对应不同类别的概率应该是不相同的。

2. 损失函数的计算公式不同。`nce_loss` 采用如下公式来计算损失函数值：

$$\text{loss} = \sum_i (y * (-\ln(G(x, y))) + (1 - y) * (-\ln(1 - G(x, y))))$$

而 `sampled_softmax_loss` 使用如下公式计算：

$$\text{loss} = \sum_i (-G(x_i, t_i) + \ln(\sum_{y \in \text{POS}_i \cup \text{NEG}_i} \exp(G(x_i, y))))$$

在上一小节中，单词向量化的例子里使用了 `sampled_softmax_loss` 损失函数对模型进行训练。这个例子同样也可以将噪声对比估计类别采样方法用于损失函数的计算进行训练，只需将原来的 `tf.nn.sampled_softmax_loss` 被替换为 `tf.nn.nce_loss`。

由以下代码可知，初始时损失函数值为 288.05871582，随着迭代次数的增加，正样本的概率密度分布和采样类别概率分布的区分度会变大。经过 10000 次训练后，损失函数值下降为 9.35187676847。需要说明的是，噪声对比估计损失函数与 `softmax` 类别采样损失函数的定义是不同的，不能仅从损失函数值来比较两者的优化性能。单词 `english` 的相似单词包括 `french`, `spanish`, `russian` 等，单词 `under` 的相似单词包括 `in`, `during`, `within` 等。将向量化的词频最高的 400 个单词进行 2 维降维处理，如图 7-2 所示。

```
Average loss at step 0 : 288.05871582
```

```
Nearest to english: darwin, whigs, malignancies, veteran,
undeniably, forgot, pounder, penzias,
```

```
Nearest to then: urbanization, abstractions, counterrevolutionary,
reestablishment, summon, defied, danelaw, emotion,
```


Nearest to any: excision, gildas, steadily, samyaksam, lemming,
kyrenia, makers, arthritis,

kyrenia, makers, arthritis,
Nearest to both: minimally, albret, rocks, rotates, litas, dijon,
pynchon, mjd,

Nearest to under: rear, mc, grays, embryonic, zirconium, compost,
phospholipids, nyc,

●●●●●

```
.....
Average loss at step 100000 : 9.35187676847
```

Average loss at step 1000000 : 9.5510707001
Nearest to english: french, spanish, russian, welsh, american,
italian, chinese, jewish,

Nearest to then: UNK, thus, when, it, five, seven, four, therefore,

Nearest to any: each, a, no, some, this, another, the, or,

Nearest to both: however, two, especially, rest, all, which, three,

many,

many,
Nearest to under: in, during, within, while, among, despite, seven,
six,

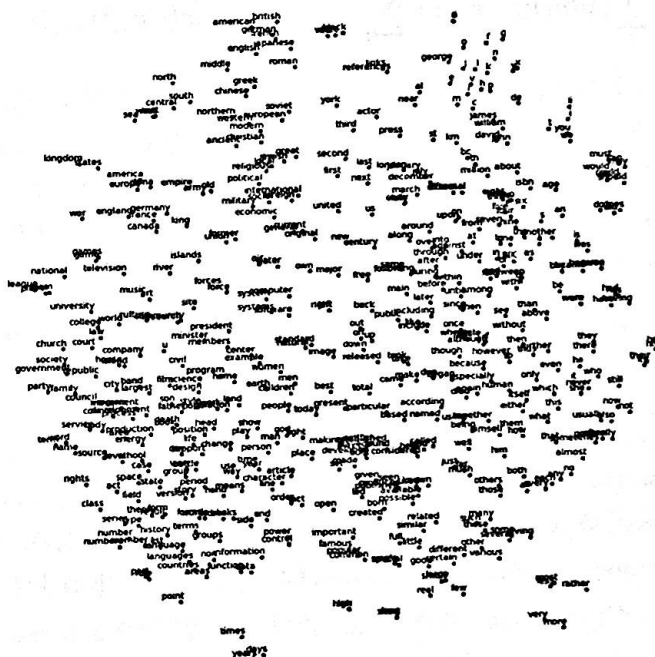


图 7-2 基于 NCE 采样包训练: 词频最高的 400 个单词向量化结果降维后的显示结果

从上述运行结果可以看出，在本例中使用nce_loss后的向量化结果并没有明显提

升。损失函数、模型结构和训练数据对最终的训练结果都会有影响。具体问题要具体分析,作为损失函数的计算方法,如表 7-1 所示,softmax和nce_loss都有自身的适用范围,在应用时需要多加思考。

7.2.3 负样本估计类别采样损失函数

负样本估计类别采样是一种简化的噪声对比估计类别采样方法。噪声对比估计强调,目标类别样本数据的概率分布和噪声数据的概率分布函数的区分度应该尽可能大。Tensorflow 中的噪声对比估计的噪声类别概率密度函数为类别采样函数。单词向量化问题的关键在于向量能否较好地反映单词间的相似或差异程度。噪声函数的概率密度函数本身并不是问题的关键。对于单词向量化这一问题,可以使用如下公式来定义损失函数,以更好地反映问题优化的优劣。

$$\sum_t (\log \sigma(\mathbf{v}_{w_o}^T \mathbf{v}_{w_l}) + \sum_{i=1}^k E_{w_i \sim p_n(w)} [\log \sigma(-\mathbf{v}_{w_i}^T \mathbf{v}_{w_l})])$$

其中 w_l 表示标签单词, w_o 表示与标签单词 w_l 对应的上下文相关单词, w_i 表示与标签单词 w_l 上下文无关的负样本单词。 \mathbf{v} 表示单词向量化结果。进行噪声对比估计时,考虑到类别子集负样本估计的损失函数没有考虑采样函数 $Q(y|x)$ 的影响。因此,从噪声对比估计类别采样方法的目标函数逼近公式推导可知,负样本估计类别采样方法下的目标函数逼近为 $\ln(E(y|x)) - \ln(Q(y|x))$ 。

值得注意的是,负样本估计方法会使目标函数逼近一个与类别采样概率分布函数 $Q(y|x)$ 相关函数。这会使最终的函数优化结果与采样分布函数的选取高度相关。这一点是与这里描述的其他所有类别采样函数都不同的。

TensorFlow 内部并没有直接实现类别采样的负样本估计损失函数。在函数 nce_loss 调用的函数 compute_sampled_logits 的注释中,可以发现当输入参数 subtract_log_q=False 时,修改 nce_loss 函数内部的输入参数即为负样本估计类别采样估计。

7.2.4 类别采样 logistic 损失函数

类别子集 logistic 损失函数是噪声对比估计类别采样方法的另一种特殊情况。它是指在噪声对比估计类别采样方法中,采样包的负样本类别集合中不包含有目标样本类

别的特殊情况。类别子集logistic损失函数要求目标类别必须为单类别集合，而负样本类别集合 S_i 则可以为多类别集合。这种训练方式相当于目标类别特征与噪声类别特征分属于两个不同的自变量区间。根据噪声对比估计中 $G(x, y)$ 的定义可知，有如下公式：

$$\begin{aligned} G(x, y) &= \log \left(\frac{\text{模型预测概率}}{\text{采样概率}(S - T)} \right) = \log \left(\frac{P(y|x)}{Q(y|x)(1 - P(y|x))} \right) \\ &= \log \left(\frac{P(y|x)}{1 - P(y|x)} \right) - \log(Q(y|x)) \end{aligned}$$

公式的第一项 $\log \left(\frac{P(y|x)}{1 - P(y|x)} \right)$ 是我们期望目标函数 $F(x, y)$ 的逼近函数。在实际的模型中，使输出层表示为 $F(x, y)$ 。但因采样类别子集的影响，输出层的实际值需要加上 $-\log(Q(y|x))$ ，此时的逻辑回归损失函数的输入为 $F(x, y) - \log(Q(y|x))$ 。并以此带入反向传播更新权重来辨别样本类别 y 是来自目标类别集合 T_i 还是来自负样本集合类别 S_i 。因反向传播误差考虑了采样概率，故目标函数 $F(x, y)$ 将逼近为 $\log \left(\frac{P(y|x)}{1 - P(y|x)} \right)$ 。

类别子集logistic损失函数可以简单地视为噪声对比估计的特例。从 TensorFlow 中函数`nce_loss`的定义可知，将默认参数`remove_accidental_hits`设置为`True`即可实现该逻辑。

7.3 小结

本章介绍了 TensorFlow 中实现的几种常用的目标函数优化算法和类别子集采样损失函数。

在 GradientDescentOptimizer、RMSPropOptimizer 和 AdamOptimizer 三种优化器中，GradientDescentOptimizer 的逻辑相对简单，所有模型参数使用相同的学习率。RMSPropOptimizer 和 AdamOptimizer 通过统计各模型参数梯度均值或方差，根据模型参数当前的梯度变化趋势，对不同模型的参数学习率进行了不同缩放。AdamOptimizer 算法考虑了训练的迭代次数，以期更好地估计梯度变化的情况进行梯度更新。

类别子集采样损失函数是一种针对大类别数据集的训练加速技术。其主要思想是在每一次分组数据训练时，通过类别采样函数获取的负样本类别。通过优化类别子集

的期望值，来逐步优化全类别集合的近似期望值。TensorFlow 中主要包括 `sampled_softmax_loss` 和 `nce_loss` 这两种类别子集采样损失函数。`sampled_softmax_loss`适用于单分类问题，整个模型被视为判别函数，且各类别间的相关性越小越好。`nce_loss`适用于多分类问题，模型被视为概率密度函数，允许不同的类别间存在一定的相关性。负样本估计与类别子集logistic损失函数都可以视为是 `nce_loss`函数的特例。其中，负样本估计忽略了类别采样函数的作用，即噪声样本概率密度的作用。仅优化样本自身概率区分度。这种方式的训练结果将极度依赖负样本的采样类别。训练结果是不稳定的。类别子集logistic损失函数则要求目标样本与噪声样本的训练特征区间不存在重叠区域。即在负样本估计类别采样类别中，不允许含有目标类别。无论是使用先验知识或进行实验测试，针对训练数据的特性选择合适的函数优化策略和损失函数都是十分必要的。

7.4 参考资料

- [1] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
- [2] Diederik P. Kingma, Jimmy Lei Ba, ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION, ICLR 2015
- [3] <http://www.jmlr.org/proceedings/paper/v9/gutmann10a/gutmann10a.pdf>
- [4] S'ebastien Jean, Kyunghyun Cho, Roland Memisevic, Yoshua Bengio, On Using Very Large Target Vocabulary for Neural Machine Translation, Cornell University Library, 2014
- [5] John Duchi, Elad Hazan, Yoram Singer, Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, Journal of Machine Learning Research 12 (2011) 2121-2159
- [6] Michael Gutmann, Aapo Hyv' arinen, Noise-contrastive estimation: A new estimation principle for unnormalized statistical models, Journal of Machine Learning Research, 2016(in processing)



结 语

纵观深度学习技术的发展历程，可以发现计算机正在经历一场认知革命。计算机从只认识一个个独立的像素值，到“看懂”一张图片中的每个物体；从只能简单存储一个个单词字符串，到“能说会道”地理解和表达出具有完整语义的句子；甚至在决策方面，以 AlphaGo 为代表的决策类算法，能够作出比人类更加明智的决定，这些无一不是令人惊讶而又兴奋的成果。

TensorFlow 凭借 Google 强大的科研和工程能力，构筑了极其强大易用的基础平台，可以让研究人员非常便利地尝试实现各种算法模型，并充分利用分布式技术提高训练效率、降低训练周期。同时，深度学习领域的研究与应用也通过 TensorFlow 这样一个平台轻松地联接在了一起，开发者能够直接将研究所得的成果转化为符合真实场景需求的应用。

深度学习如今仍然是处在快速发展过程中的一个技术方向，研究实验中所取得的各种成绩并不会成为终点。在高性能计算硬件、大数据、编程框架等多种技术的支持下，相信在不远的将来，深度学习终将成为改变人类生活方式的重要基石。

模型发展出更深的结构、更大的规模也是必然趋势。更何况实验已经证明，更大的模型能够达到更好的表现。以图像识别为例，拥有 8 个参数层的 AlexNet 在 ImageNet 大规模视觉识别挑战（ILSVRC）中将图像分类的错误率降低到 15.3%，而有 19 个参数层的 VGGNet 的分类错误率只有 7.3%，现今最大、最深的 ResNet 有超过 150 个参数层，分类错误率仅为惊人的 3.57%，这已经是超越人类能力的成绩。

可以预见的是，深度学习在未来将能处理越来越复杂的任务，也注定会在我们日常生活中扮演越来越重要的角色。

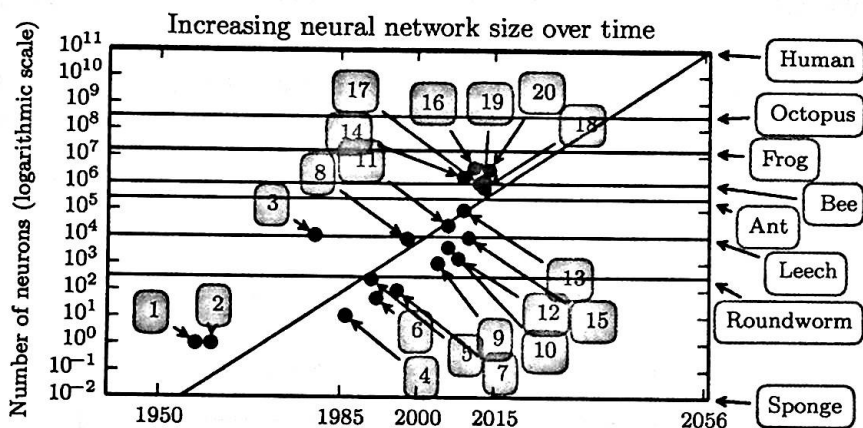


图 1-6 自从隐层结构提出以来，神经网络的神经元数量每 2.4 年翻一番。图中各编号代表的分别为：1. 感知机（Perceptron）；2. 自适应线性单元（Adaptive linear element）；3. 神经认知机（neocognitron）；4. 早期反向传播网络；5. 用于语音识别的递归神经网络；6. 用于语音识别的多层感知机；7. sigmoid 置信网络；8. LeNet-5；9. 回声状态网络（Echo state Network）；10. 深度置信网络（Deep belief network）；11. GPU 加速的卷积网络；12. 深度玻尔兹曼机（Deep Boltzmann machine）；13. GPU 加速的深度置信网络；14. 非监督卷积网络；15. GPU 加速的多层感知机；16. OMP-1 网络；17. 分布式自动编码器（Distributed autoencoder）；18. 多 GPU 加速的卷积网络；19. COTS HPC 非监督卷积网络；20. GoogLeNet

1.3 参考资料

- [1] 本部分内容参考并翻译自：Ian Goodfellow and Yoshua Bengio and Aaron Courville. "Deep learning." An MIT Press book in preparation. Draft chapters

available at <http://deeplearningbook.org/> (2016).

- [2] Bengio Y, Courville A, Vincent P. "Representation Learning: A Review and New Perspectives". arXiv:1206.5538v3.
- [3] Geoffrey Hinton, Yoshua Bengio, Yann LeCun. "Deep Learning". NIPS 2015 Tutorial.
- [4] 周志华, 《机器学习》.
- [5] Yann LeCun. "Deep Learning". ICML 2013 Tutorial.
- [6] Bengio, Yoshua, and Yann LeCun. "Scaling learning algorithms towards AI." Large-scale kernel machines 34.5 (2007).